ANALYZING AND MODELLING WEB SERVER BASED SYSTEMS

Pasindu Nivanthaka Tennage

188012C

Degree of Master of Science

Department of Computer Science and Engineering

University of Moratuwa
Sri Lanka

July 2020

ANALYZING AND MODELLING WEB SERVER BASED SYSTEMS

Pasindu Nivanthaka Tennage

188012C

Thesis submitted in partial Fulfillment of the Requirements for the Degree Master of Science

Department of Computer Science and Engineering

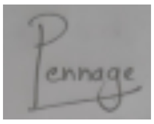
University of Moratuwa
Sri Lanka

July 2020

Declaration

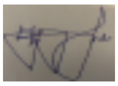
"I declare that this is my own work and this thesis does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my thesis, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books)".

Signatures:  Date: 2020-06-15

The above candidate has carried out research for the Masters thesis under my supervision.

Signature (Sanath Jayasena):  Date: 2020-06-15

Signature (Malith Jayasinghe):  Date: 2020-06-15

Abstract

Server based systems are widely used in modern computer systems. Understanding the performance of web server based systems, under different conditions is important. This requires a step by step approach that includes modelling, designing, implementing, performance testing and analyzing of results. In this research, we aim at characterizing the web server systems under different configurations. We present a summary of prevalent server architectures, provide a systematic approach for performance testing, and present a novel open source Python library for latency analysis. We experiment on existing server architectures, and propose eight new server architectures. Our analysis shows that under different conditions the new architectures outperform the existing architectures. Moreover we do an extensive tail latency analysis of Java microservices.

Key words: Server architectures, tail index, performance, latency, throughput, web

Acknowledgements

I would like to acknowledge with greatest gratitude the help and guidance I received to conduct this research, from these respected persons. I would like to show my deepest gratitude to project supervisors Professor Sanath Jayasena and Dr Malith Jayasinghe for the thorough guidance and the consistent assistance I received throughout this research. My gratitude goes to Dr Srinath Perera for guiding me in the research through numerous consultations.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Description
CCDF	Complementary Cumulative Distribution Function
CDF	Cumulative Distribution Function
DES	Discrete Event Simulation
HTTP	HyperText Transfer Protocol
LLCD	Log Log Complementary Graphs
PDF	Probability Density Function
REST	Representational State Transfer
RPC	Remote Procedure Call
SEDA	Staged Event-driven Architecture
SOAP	Simple Object Access Protocol
UDDI	Universal Description, Discovery, and Integration
URI	Uniform Resource Identifier
WSDL	Web Services Description Language
XML-RPC	XML Based RPC

1. INTRODUCTION

Server based systems are widely used in modern computer systems. Hence, understanding the performance of web server based systems, under different conditions is essential. This requires a systematic approach that consists of modelling, designing, implementing, performance testing and analysing the results. In this research, we aim at characterizing the web server systems under different configurations.

Server based systems cover a wide range of applications that can be categorized into different architectures. Hence it is important to first understand different architectural styles such as monolithic and service oriented architecture. We first present a case study of existing web architectures, such as monolithic, and microservices.

Second we provide a systematic approach to characterize the server based system performance. We explore the benchmarks, workload generation, theoretical models, tools and measurement technologies. There, we present a novel open source Python library for latency analysis.

Third, we discuss the existing server architectures, Blocking, NIO, NIO2 and Staged event driven architecture, and message passing architectures, Queue, Disruptor and Actors. Then we propose 12 web server architectures, eight of which are novel, which are combinations of above server architectures and message passing architectures. We perform an extensive analysis of the 12 architectures and show that the novel architectures outperform the existing architectures for some use cases.

High tail latency is an important area of systems research. Many prior work have attempted to characterize and mitigate the long tail latency values of different types of systems. However, tail latency characteristics of microservices is still an unknown area with little to no existing prior works. As the fourth part of this thesis, we explore the tail latency characteristics of microservices. Our findings and conclusions are published in Tennage et al. [89], hence we only provide the references in this thesis. An interested reader can refer to our original publication [89] for complete details.

We then focus on scalability characteristics of web servers. We first discuss the theoretical models of scalability, Amdahl’s law and Universal Scalability law. Then we perform an extensive analysis of WSO2 Enterprise Integrator scalability using Universal scalability law as the model.

Discrete event simulation is a technique that can be used to model server based systems, without performing extensive tests. In this approach a server is simulated in a virtual environment. Due to its flexibility and cost effectiveness, discrete event simulation has gathered a wide recognition in workload characterization. As the seventh part of this research we explore how to use discrete event simulation for modelling web server systems. We propose a novel method to model the closed system model using discrete event simulation and study the impact of number of cores and concurrency on performance.

Load balancing is a popular scaling technique for web servers. However, there is a myth that load balancing always improves performance. To address this problem, we study the impact of load balancing for web server systems. Our analysis shows that blindly adding multiple resources using a load balancer does not improve the performance and explain in detail the performance impact of load balancing.

Following are the major contributions of this research.

1. A novel open source Python library for workload characterization
2. A systematic approach for performance testing of web servers
3. Propose eight new server architectures
4. Hardware, Software implications for server performance
5. Tail latency analysis of microservices

6. Identifying the scalability characteristics of middleware using Universal law of scalability
7. A novel approach to model web server closed system performance using discrete event simulation
8. Identifying the impact of load balancing for server based systems.

1.1. Research Problem

1.1.1 Motivation and overview

Server based systems are widely used in modern computer systems. High performance is a requirement of server based systems to ensure the service level agreements. Hence, it is important to understand the performance behaviours of server based systems. We explore the performance characteristics of server based systems for different architectures and configurations.

1.1.2 Problem statement

In this research we focus on characterizing the performance of server based systems.

1.2. Research Objectives

Objective of this research is to identify the performance behaviour of server based systems. Our specific aims include

1. Implementing an open source Python library to analyse latency.
2. Design and implement eight new server architectures.
3. Tail latency analysis of microservices.
4. Application of universal scalability law to analyse middleware scalability.
5. Implementing closed system performance tests using discrete event simulation.
6. Explore the impact of load balancing for server based systems.

2. RELATED WORK

2.1 Web Services

Web Services are widely adopted in modern computer systems. Web services are based on Hypertext Transfer Protocol (HTTP). Web services replaced Remote Procedure Call (RPC) technology which was the state of the art method before web services. Using explicit message passing techniques, web services provide communication between nodes [1].

XML based RPC adapts RPC technologies to web services [1]. XML-RPC uses POST requests (HTTP) for executing procedure calls. Though XML based RPC (XML-RPC) uses some of the features available in HTTP protocol, there is a considerable number of features of HTTP that are not used in XML-RPC [1] [2] .

SOAP is an improved version of XML-RPC, which addresses some limitations inherent in XML-RPC. SOAP specification defines the message format and methods for the exchange of messages between services [3]. There exist several variations of SOAP which are labelled as WS-*. These extensions address the additional features of SOAP including security and service orchestration [1].

Web services description language (WSDL) is a standard for inter service communication [4]. Unlike RPC based protocols, WSDL uses service descriptions that are in XML format.

Universal Description, Discovery and Integration (UDDI) is another method in web services, which was originally used to provide registry functions [5]. Due to emerging new technologies, which we discuss in the later sections, UDDI's importance is almost lost.

Although SOAP, WSDL and UDDI have gained much popularity in web services, it does not cover the full spectrum of web. Though HTTP can be used as an application level protocol, SOAP and WSDL use HTTP only as a communication protocol. Representational State Transfer (REST) addresses these issues in WSDL and SOAP [6]. REST incorporates most of the fundamentals of the World Wide Web. REST architectural style specifies a wide range of HTTP features.

There are several features of resource oriented architecture.

1. Client-server architecture: Separation of responsibilities is the major aspect of this architecture. Client Server architecture proposes to evolve client and server as two independent systems. This enables decoupled code which is easy to develop.

2. Statelessness: Statelessness property of resource oriented architecture promotes scalability and reliability.

3. None shared cache: Helps to reduce latency and improves efficiency by reducing misses.

4. Layered system: Layer is a group of reusable components that are reusable in similar circumstances. Using a layered system enables to balance the server workload among many nodes.

5. Code-on-Demand: This is a component of resource oriented architecture which provides extensible deployment.

6. Uniform interface: This enables to use and access resources using HTTP methods such as GET and POST.

2.2 Concurrency

Running multiple activities at the same time is defined as the concurrency [7]. Though executed in parallel, the tasks may interact among them. Concurrency in computer systems is available in many forms such as multiple cores and single core–multiple threads. When implemented correctly, concurrency can reduce the latency and improve the level of throughput.

Web applications are also inherently concurrent. Both the application server and the web server should handle the concurrency issues. Roy et al. [7] states four main approaches for programming concurrency, concurrency based on shared state, concurrency based on message passing, concurrency based on declaration and no concurrency (sequential program execution).

Concurrency in distributed systems has a set of inherent challenges as shown by Arnon et al. [9]. Network, infrastructure, latency, topology, transport and bandwidth are shown as the major challenges in developing concurrent applications in a distributed environment.

Ghosh et al. [10] have evaluated different programming languages such as Java and C++ with respect to their adoption in distributed systems. They have shown the limitations of existing languages and suggested different approaches to mitigate them. Hence novel languages such as Ballerina [11] are emerging for distributed application development.

2.3 Scalability

Scalability aims at handling dynamic workload by changing the deployment, either hardware or software. Scalability becomes a requirement that necessitates the usage of a distributed system.

There are two methods of scaling; 1. Vertical and 2. Horizontal. In vertical scaling, more resources (such as RAM and CPU cores) are added to a single computing unit.

Hence, the node can handle more tasks. In contrast, more nodes are added in the horizontal scaling approach. Horizontal scaling requires specific treatment in the application implementation [1].

2.4 Web Server Architectures

Web server architecture defines how the input output of requests are handled and how different threads are allocated. Web servers are designed with the aim of maximizing the performance while using a minimum number of resources [1]. Kegel et al. [12] have highlighted the need for improved server architectures.

Thread based and event based are the most widely used web server architectures, from which most other architectures are derived. More sophisticated variants have emerged which combine these two approaches [13] [14] [15].

The thread-based/process based approach associates each incoming connection with a separate thread/process (synchronous blocking I/O). Thread/process based approach is supported by many programming languages.

Process per connection is the first attempt on building web server architectures [16]. In this approach a separate process is assigned to each client request. Though this model is easy to implement, it has been shown that this method cannot support very high concurrencies. Processes are heavy weight and require a lot of computer resources. When the concurrency increases, the server cannot fork processes than a maximum that is imposed by the underlying hardware and operating system.

Thread per request architecture is the successor of process per connection approach [16]. In this architecture, a thread is assigned for each client. Compared to processes, threads are lightweight. Hence this architecture scales more than the process per connection architecture. However, when the concurrency increases beyond a threshold, the underlying hardware cannot support a large amount of threads. To address this issue, bounded thread pool architecture was proposed [16]. In this approach, a predefined thread pool is used. If the number of clients is greater than the size of the thread pool, the additional requests queue up.

However, thread per connection approach does not provide good quality of service at high concurrency levels, due to overheads of context switching and stack management. Non-blocking IO which is also called as event driven architecture addresses these issues by handling multiple connections using a single thread [17]. In this approach, a single thread listens to all the events (accept, read and write events) and handles them in a non-blocking fashion [17]. This method scales well until all the operations are non-blocking.

Welsh et al. [15] have combined these two approaches; thread pool and event based, and proposed an architecture called staged event driven architecture (SEDA). In this architecture, a stage is defined as an execution unit which has a thread pool. Request processing is done at several stages, for example the first stage accepts the connections, and the second stage reads the sockets. This enables to improve the non-blocking IO performance. A later study [18] has argued that SEDA performance suffers when the workload is low, due to its implicit restrictions on stages.

2.5 Message Passing Architectures

Message passing architectures account for the mechanism that is used to pass a message from one thread/process to another. There exist three widely used message passing architectures.

1. Queue

   Queue is a data structure that is commonly used in programming. Queue support operations such as enqueue and dequeue. Queue follows First-In-First-Out methodology. Most thread pool based architectures use queues to enqueue new runnable items.

2. Disruptor

   LMAX [19] addresses some fundamental limits with conventional queues, such as latency costs. Disruptor introduces a ring buffer that can be used instead of queues. Disruptor aims at reducing cache misses by pre allocating the objects in the ring buffer.

3. Actor

   Actor model is a method of decoupling different entities. It provides explicit asynchronous message passing techniques using mail boxes [20] [21]. Actors can perform the following actions.

   - Send a message to another actor.
   - Create new actors.
   - Change actor's own internal behaviour

2.6 Microservices

Please refer to the literature review section of our previous publication [89].

2.7 Summary

In this section we first discussed the evaluation of web services. Then we presented a summary of prevalent concurrency architectures in web servers. Moreover, we discussed existing server architectures while highlighting their pros and cons. Finally we reviewed literature on Microservices; the architecture, their wide adoption and the importance of microservices performance characterization.

3. SERVER ARCHITECTURES

3.1 Introduction

Server architectures have evolved rapidly over the last three decades. In this research, several server architectures, including monolithic, and microservices are discussed in terms of their performance.

3.2 Client Server Paradigm

Client-Server paradigm is the first widely adopted architecture for developing systems that are distributed [40]. In this model, two programs communicate with each other, client and the server. Client initiates the communication using a request which is received by the server.

In this architecture, the server is the component which provides all the resources requested by the client [41]. This architecture was initially proposed as an all in one architecture. Later on, due to complexities of all in one method, tired architecture was proposed. There, the server is broken into n-tiers. Three-tier architecture is one of the most popular approaches.

3.3 Mobile Agents

Mobile agents architecture addresses the limitations of the client server architecture [23]. In this architecture, an agent can move from one node to another with its code and data structures [42]. Mobile agents have three main components: owner, locations visited and the adversary. Ismail et al. [40] have shown that depending on the visited nodes an agent can evolve on its own.

3.4 Service Oriented Architecture

Service Oriented Architecture (SOA) is the most successful alternative for client server architecture [43]. SOA promotes less coupling among services [40]. SOA defines explicit boundaries between different services.

Enterprise Service Bus (ESB) plays an important role in SOA. Each client request is first handled by the ESB. ESB specifies the relevant service that can serve the request using its registry [44].

3.5 Microservices

Microservices is an architectural style that was proposed recently [45] [46] [47]. Microservices aims at building loosely coupled services which are easy to deploy. Though argued as an extension of SOA, microservices is a bottom up approach that addresses the requirements of distributed systems.

Microservices comprises a set of small services which can run independently. Communication between services is strictly using message passing. Though its performance suffers compared to monolithic architecture, its improved usability, flexibility and ease of deploying have made it the de facto method of distributed computing.

4. PERFORMANCE ENGINEERING

4.1 Introduction

Workload characterization is a sub field of distributed computing, which tries to implement systems, collect data, and analyse to draw conclusions. It requires both implementation skills, and analytical skills to perform a good workload characterization.

Workload characterization is important in many different aspects.

1. Helps to identify current performance of your system, which can be used for service level agreements of commercial applications.

2. Helps to identify the optimum setup of a system.

3. Helps to identify the performance bottlenecks in a system.

Workload characterization has emerged as a systematic procedure, with several best practices of its own. These best practices are as follows.

1. Selecting the correct benchmark

   Selecting the correct benchmark is the most crucial and essential part in workload characterization. For example, if one intends to do a performance analysis of microservices, it is required to select a standard microservices benchmark like Socks Shop. A benchmark should have the following set of characteristics.

   a. Should represent a read world application
   b. Should be able to handle real world workload
   c. Should adhere to the standard best practices of the architecture

2. Selecting the correct model

   There are three main models that can be used for performance testing.

   a. Open model
   b. Closed model

c. Half open model

Each of these models are suitable for different purposes, for example, the open model is suitable for tail latency analysis, whereas the closed system model is used for analysing the maximum sustainable throughput of a server. In the following sections, each of these models are described in depth.

3. Selecting the correct workload generation tools and workload

   For each type of model mentioned above, there are corresponding workload generation tools, for example Apache JMeter [48] is a widely used tool for closed model workload generation.

   Also, there are two types of workloads, synthetic and real. Synthetic workloads mean the work that is generated synthetically, for example sending requests to a server using JMeter using a specified level concurrency. In contrast, there are standard real workloads such as 98 World cup dataset, which are accepted as standard for a web server workload.

4. Standard performance metrics

   There are a set of performance metrics that are accepted both in industry and academia; average latency, throughput, 99 percentile latency. Selecting the right set of performance metrics for the intended work is very crucial. In the following sections, an introduction to most important performance metrics are given.

4.2 Benchmarks

In this section, the popular benchmarks that are used for server workload characterization are listed.

1. Microservices Benchmarks
   a. Acme Air
   b. Spring Cloud demo apps

   c. Socks Shop
   d. MusicStore

2. Monolithic Server Benchmarks
   a. SPECweb 2009
   b. TPC – C
   c. TPC – W
   d. SpecjEnterprise 2010

4.3 Workload Generation

Once a suitable benchmark has been selected, a suitable workload generation mechanism should be used. There are two main workload generation mechanisms.

4.3.1 Real workloads

In real workloads, real user traffic that are extracted using HTTP Logs are replayed. This enables to simulate the system as in a real world deployment. For a given application, it is possible to collect HTTP logs over a period of time, and then use these logs to replay the traffic. Also, there are standard workloads that are widely accepted in literature such as the 1998 World Soccer Cup dataset.

However there are several bottlenecks of using real workloads.

1. Takes a long time to collect data.
2. Cannot collect data for different configurations in a production environment.
3. There may not be standard workloads for newer architectures such as microservices.

4.3.2 Synthetic workloads

To address the problems associated with real workloads, workload characterization experts use synthetic workloads; workloads that are generated artificially. This approach has several advantages.

1. Can do perform test in a reasonable time
2. In a case where a huge number of events (requests) are needed to characterize, this is the only option, since the earlier approach of real workloads does not allow to get many requests.
3. Can change the server parameters and perform the tests.

4.4 Performance Models

Workload generators are classified based on a performance model [54]. Performance models impact the performance numbers, hence they are a major concern in performance engineering.

4.4.1 Open loop model

In the open loop model, a client sends a request to a server and then leaves the system immediately. A typical Google search operation can be taken as an example of this model. There, a user sends a request to the Google server and with a high priority leaves the Google server by going to another web site.

4.4.2 Closed loop model

In the closed loop model, a client repeatedly sends requests to the server. First the client sends a request to the server. Upon receiving the request, the client sends another request to the same server.

4.4.3 Half open model

None of the above two models are representative of real traffic. In a real client server interaction, a client first sends a request to the server, and then for some time acts like in a closed system loop, and once the intended work is done leaves the system. This model is called the half open model.

4.5 Tools

There are several standard tools that emulate the user traffic. Table 4.1 below summarizes the workload generation tools.

In this research we use Apache JMeter for all the workload generation tasks, and we incorporate the closed system model. We use this model since we are interested in characterizing the performance of the server under server's peak sustainable throughput.

JMeter

We use the load testing tool, JMeter to simulate the virtual users. Figure 4.1 depicts the experimental setup for load testing. At a given concurrency level, JMeter client sends the same request to the configured endpoint (address of the server). For example, if we use a concurrency of 100 users, JMeter starts 100 threads and starts sending requests to the Server. Upon receiving a request, the server processes the request and sends the response back to the JMeter client. Upon receiving the response from the server, each JMeter thread sends the next request (a user can specify a think time). It is assumed that JMeter client has enough hardware resources to handle the given concurrency level. If the concurrency level is greater than the maximum capacity of the machine, a distributed JMeter setup should be used to distribute the load among many JMeter nodes. By collecting the JTL file (saved in the JMeter client), latency values for each request are collected.

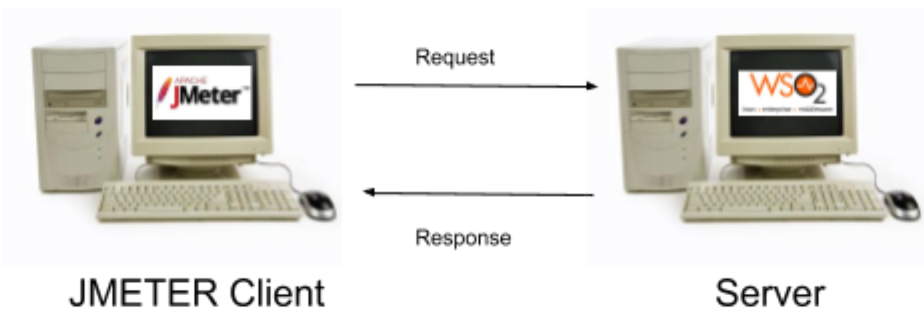


Figure 4.1: JMeter Experimental Setup

Table 4.1 Workload Generation Tools

Benchmark Type	Examples	Workload Model
Model based web workload generation	Surge, WaspClient, Geist	Closed
Playback mechanisms for HTTP request streams	MS web application stress tool	Open
Proxy server benchmarks	Wisconsin proxy benchmark	Closed
Database benchmark for e- commerce workloads	TPC-W	Closed
Auction website benchmark	Rubis	Closed
Online bulletin benchmark	Rubbos	Closed
Database benchmark for online transaction processing	TPC-C	Closed
Model based packet level web traffic generators	IPB	Closed
Mail server benchmark	SpecMail	Open
Java client server benchmark	SPECJ2EE	Open
Web authentication and authorization	AuthMark	Closed

Network file servers	NetBench	Closed
Streaming media service	Medisyn	Open

4.6 Performance Measurement

Once a suitable benchmark and a workload generation tool are selected, comprehensive performance tests are run. Then, using the results, it is possible to extract different performance measurements that are useful for analysing the performance. In this research we extract performance measurements using JMeter reports, garbage collection logs, Linux SAR reports and Linux PERF reports.

4.7 Latency Analysis Methods

Latency values are the most important dependent variable in a performance test. Analysing latency should be done with care. There exist several mathematical methods of latency characterization. Yet, there is no implementation for these methods. In this research, we implement a novel python library that helps to mathematically analyse the latency values [55]. In the following sections, each method is described in detail.

4.7.1 Average latency

Average latency gives an overview of the overall performance of the system. Though average latency is not capable of revealing the extreme values of the dataset (also known as tail latency values), it is still useful to get a general idea about the performance.

4.7.2 Latency percentiles

As we mentioned earlier, the average latency does not reflect the impact of extreme values. Hence we need robust figures to analyse the extreme latency values. Furthermore, when writing commercial applications, it is vital to make sure that these extreme values are within the agreed values in the service level agreements (SLA). Hence, there exists a significant importance to identify these extreme latency values.

In general, most computer workload latency distributions are right skewed, meaning that there exist extreme values. Hence, higher order latency percentile values are used to capture these extreme latency values.

Percentile of a dataset is the value which is the lowest among the values which are higher than a given fraction of values. For instance, assume a sample dataset of 100 values, organized in ascending order. Then the value at 90th position is greater than 90% of the values in this dataset; hence it is the 90th percentile of the dataset.

4.7.3 Distribution analysis

Computer Workload latency values come from continuous distributions. Hence they can take any value in a given range. Hence there exists an underlying distribution for the latency values observed in a computer system workload. Identifying this underlying distribution of a given set of latency values helps us to characterize the system better. This enables us to synthetically generate the workload and experiment further on the computer system. In this section, first the most basic form of distribution analysis, histograms and probability density functions are presented. Then methods on how to check whether a given latency distribution adheres to a theoretical continuous distribution are explored. Maximum Likelihood Estimation is used to fit the observed latency values to a given theoretical distribution. Then goodness of fit tests are used to identify how well theoretical distributions characterize observed latency distributions. Three most widely used goodness of fit tests, Quantile-Quantile Plot (Q-Q plot), K-S test and Chi Squared test are presented.

Histogram

Histograms are the most basic method of characterizing latency values. It simply shows the frequency of different values. Histograms are useful when analysing a relatively small set of latency values (less than 100), and are more applicable when the range of the data (maximum value - minimum value) is relatively small. When the range is large, logarithmic binning should be used. To have more meaningful representation of data, a technique called binning is used. Dividing the range into small regions is meant by binning. For an example, if there is a set of latency values in the range (0, 100), it is more meaningful to use a bin size of 10, such that values in a given bin (for example values in the range (0, 10) are treated as the same. Figure 4.2 illustrates a sample histogram obtained using a normal distribution with mean 1 and standard deviation 0.5.

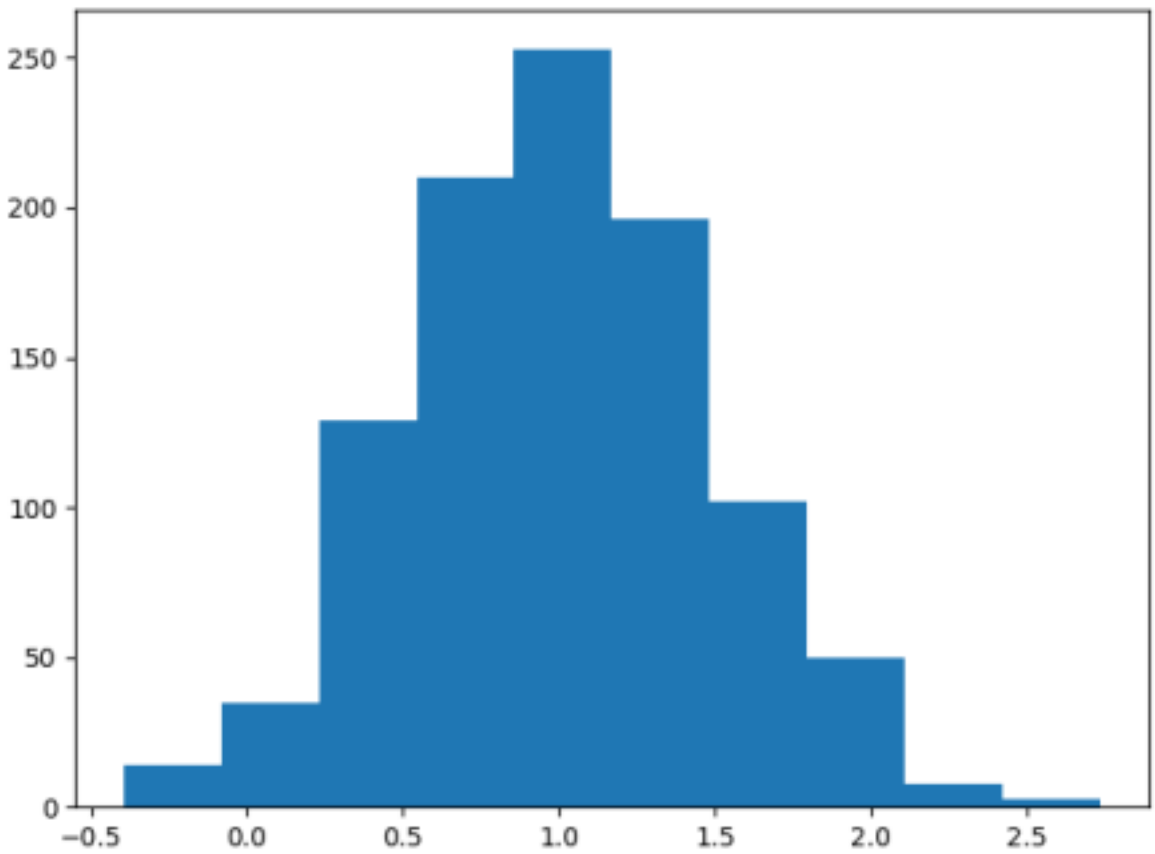


Figure 4.2: Histogram

Probability density function

Probability Density Function (PDF) PDF calculates the probability of occurring a value in a given range. Equation (4.1) denotes the pdf equation.

$$Pr(x <= X < x + \delta x) = f(x)\delta x \qquad (4.1)$$

The pdf f, is not a probability. At any given x value, it has a value of 0. By multiplying by the range, it can be converted to a probability. Probability density function of a set of latency values is calculated using Kernel Density Estimation. Let (x_1, x_2, …, x_n) be an independent and univariate sample drawn from an unknown distribution with an unknown density function f. Then its kernel density estimator is calculated using (4.2).

$$f(x) = 1/n * \Sigma k(x - xi) \qquad (4.2)$$

k is the kernel function which is non-negative Figure 4.3 depicts a sample kernel density estimation.

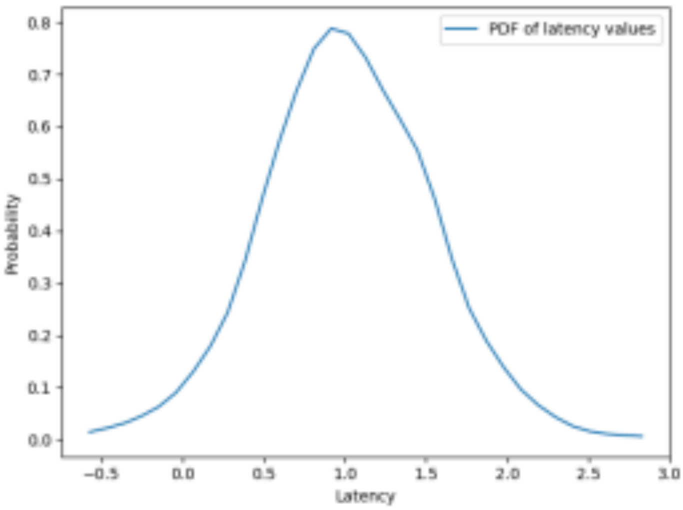


Figure 4.3: Probability Density Function

Cumulative distribution function

Cumulative distribution function (CDF) F is defined as the probability that a set of latency values is smaller than or equal to a given latency, as denoted in (4.3)

$$F(x) = Pr(X <= x) \quad \text{.......(4.3)}$$

Since latency values are continuous, the CDF is obtained by integrating the PDF, as denoted in (4.4).

$$F(x) = \int_{-\infty}^{x} f(t)dt \quad \text{.......(4.4)}$$

In equation 4.4, f denotes the PDF whereas F denotes the CDF. Figure 4.4 below depicts the CDF obtained for the sample dataset obtained from random number generation using Pareto distribution with tail index 1.

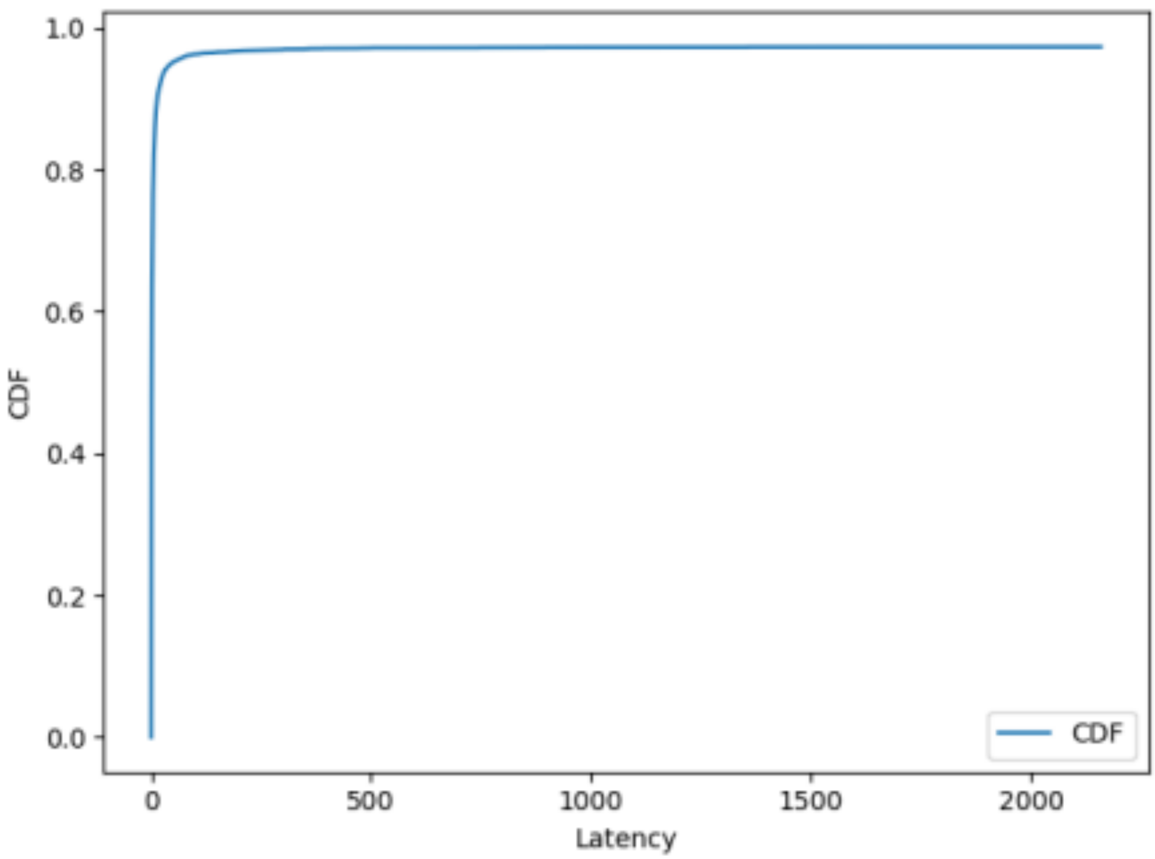


Figure 4.4: Cumulative Distribution Function

4.7.4 Theoretical distributions

Computer Systems' workloads have specific distributions. These distributions sometimes follow known theoretical distributions such as the Pareto Distribution and Exponential Distribution.

Understanding the underlying distribution of the latency values helps us characterize the system better. Moreover, it paves us way to use Computer Simulation for capacity planning.

Theoretical Distributions have three types of parameters.

1. Shape Parameters: denotes the shape of the distribution
2. Location Parameters: denotes the value around which the distribution is located (for example mean value in the normal distribution)
3. Scale Parameters: denotes the amount the distribution is spread out (for example the standard deviation in the normal distribution)

Hence, the first step of checking latency distributions against the standard continuous distributions is to identify these parameters. In this research, we focus on the most widely used parameter estimation method, Maximum Likelihood estimation. Once parameters are calculated, it is then needed to know how good latency distribution fits with the theoretical distribution with the calculated parameters. For that purpose, three widely used Goodness of fit tests, Q-Q plot, K-S test and χ2 test are used.

Maximum likelihood parameter estimation

Maximum likelihood method retrieves the parameters that maximizes the opportunity of observing the given data. The likelihood function is the probability of observing a set of values given that they fit to a distribution. If parameter θ defines the distribution, equation 4.5 gives the maximum likelihood estimation for a set of latency values, $x_1, \ldots, x_n$.

$$L(\Theta|x1.....xn) = \Pi f(xi|\Theta) \quad (4.5)$$

Once the set of parameters in the theoretical distribution are calculated, then it is possible to draw the theoretical distribution with the calculated maximum likelihood parameters. Figure 4.5 depicts the calculated Pareto distribution.

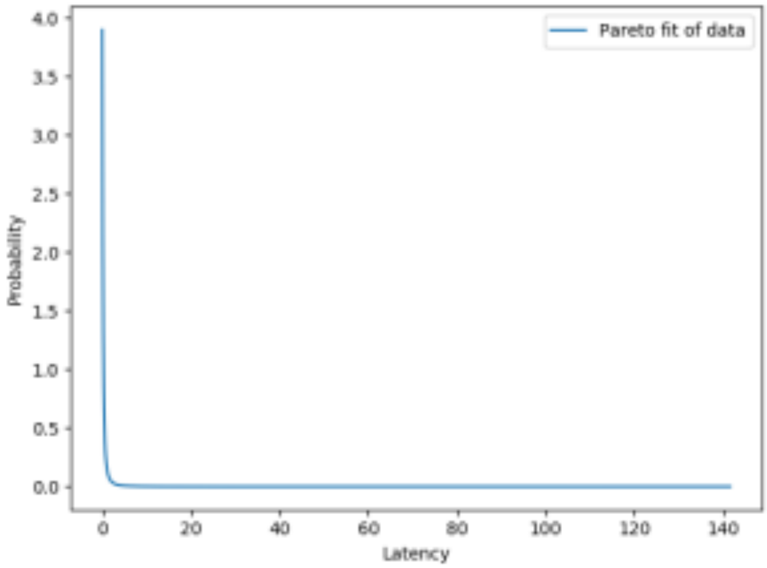


Figure 4.5: Maximum likelihood Pareto fit of data

Goodness of fit tests

Once the parameters are estimated using maximum likelihood estimation, we should test how good our parameter approximation is. There exists three main methods of testing the goodness of fit tests; Quantile-Quantile plots, Kolmogorov and Smirnov test (K-S test), and $\chi 2$ test.

Quantile-quantile plot

Quantile-Quantile plot is a method of comparing distributions. The percentiles of one distribution is plotted against the respective percentiles of the other distribution. If the observed distribution follows the theoretical distribution, the percentiles should lead to a straight line with slope one. Except the graphical plot, this method does not provide any quantitative value indicating the goodness of fit.

Kolmogorov and smirnov test

The Kolmogorov and Smirnov test calculates the maximum distance between the CDF of the theoretical distribution and empirical distribution. If the samples follow the theoretical distribution F(x), then

$$Pr(\lim_{n \to 0} |F(x) - Fn(x)| = 0) = 1 \qquad \text{...(4.6)}$$

$$Dn = sup|F(x) - Fn(x)| \qquad \text{..(4.7)}$$

With n as the number of data points, Fn(x) is a unit step function. Hence,

$$Dn = max(|i/n - F(Xi)|, |F(xi) - (i-1)/n|) \qquad \text{.........(4.8)}$$

If the D_n is small enough (with respect to the chosen significant level), the empirical distribution follows the theoretical distribution.

When deciding whether the latency distribution fits the theoretical distribution of interest, we check the p value returned by the Kolmogorov and Smirnov test. If this p value is greater than our pre specified significance level (0.05 in practice), we say that this is a good fit.

In reality, we don’t exactly know what the underlying theoretical distribution our observed latency values follow. In that case, we should check for all possible theoretical distributions and then select the theoretical distribution which closely matches with our observed latency distribution.

χ2 method

In χ2 method, random samples are drawn from the theoretical distribution of interest. Then these samples are compared against the observed samples. χ2 test statistic is computed as in (4.9).

$$\chi^2 = \Sigma(Oi - Ei)^2/Ei \quad \text{...(4.9)}$$

4.7.5 Long tail distribution analysis

In most cases, Computer Workloads are long tailed, meaning that there exist a small fraction of latency values that are relatively large compared to the mode and average latency. Hence, there exists a significant importance in characterizing the long tail nature of latency values. In this section, we first present two properties of long tail distributions, power law behaviour and mass count disparity. We then present a method to discriminate between heavy tailed distributions and non-heavy tailed distributions. Finally, we present three methods to calculate the tail index, which is the most widely used statistical method of characterizing long tailed distributions.

.

Properties of long tail distributions

Power law behaviour

The long tailed distributions can be characterized using power law equation (4.10).

$$F(x) = Pr(X > x) \propto x^{-a} \quad \text{..(4.10)}$$

Pr*(X>x)* is the survival function, which is (1 - F(x)) where F(x) is the empirical cumulative distribution function. The exponent ‘a’ is called the tail index, which determines the tail behaviour of the distribution. Lower the value of a, higher the tail of the distribution (chance of observing a small fraction of very high latency values becomes high).

Mass count disparity

Mass-count disparity is a property of long-tailed distributions. This means a typical item is short, but a typical item of total test belongs to an item whose length is very large. Mass count disparity is characterized by comparing mass distribution with the count distribution.

Count distribution is the Cumulative Distribution Function. Assuming a probability density function, f(x) mass distribution can be expressed as in (4.11).

$$F(x) = \int_0^x x' f(x')dx' / \int_0^\infty x' f(x')dx' \qquad \text{..........................(4.11)}$$

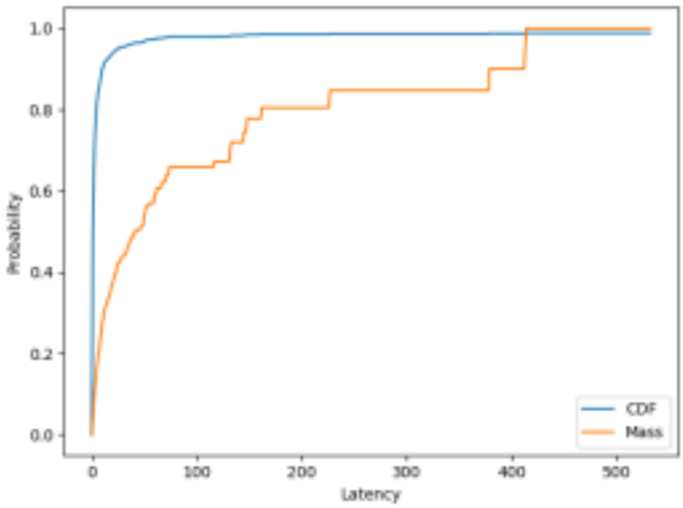


Figure 4.6: Mass Count Disparity

Mass count disparity provides four main quantitative measurements to identify the long tail behaviour of data; joint ratio, N half, W half and Gini coefficient.

Joint ratio is the value ‘p’, such that p% of the items account for (100 − p) % of the mass, whereas (100 − p)% of the items account for p% of the mass.

$$P = 100 * Fm(x), Fc(x) = 1 - Fm(x)$$ ....(4.12)

N-1/2 and W-1/2 are two generalizations of 50/0 principle. 50/0 principle states that 50 percent of the items account for a negligible mass.

$$N1/2 = 100(1 - Fc(x)), Fm(x) = 0.5$$ ....(4.13)

$$W1/2 = 100Fm(x), Fc(x) = 0.5$$ ....(4.14)

Smaller the values of N-1/2 and W-1/2, heavier the tail of the dataset becomes.

Gini coefficient is another measurement of estimating the long tailedness of data, which uses mass distribution and count distribution. Gini coefficient uses Lorenz curve, which is the percentile-percentile plot of mass distribution and count distribution.

Gini coefficient computes the inequality between mass distribution and count distribution. Gini coefficient is the ratio of the area between the equality line and the Lorenz curve, and all the area below the equality line (figure 4.7). Gini coefficient varies in the range (0, 1).

$$G = 2\int_0^1 (x - L(x))dx$$ ....(4.15)

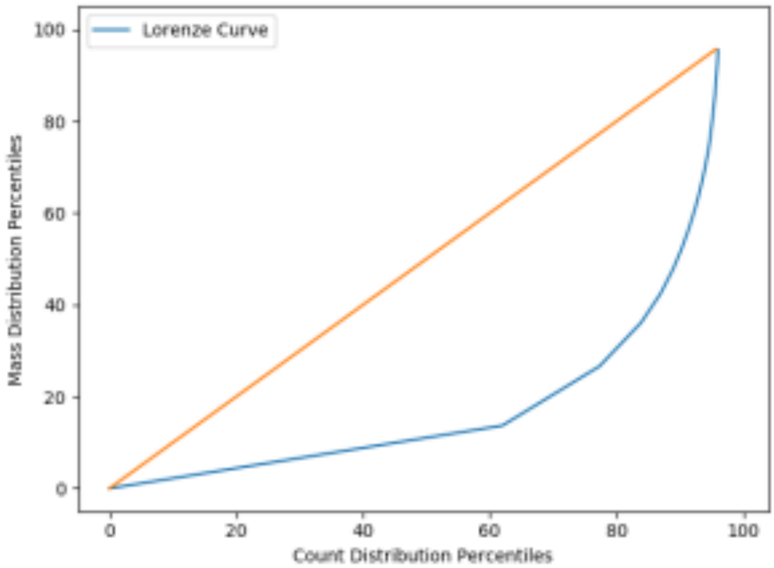


Figure 4.7: Lorenz Curve

Heavy tailed distributions

Heavy tailed distributions are a subset of long tail distributions with some specific characteristics. Heavy tailed distributions have the following three properties.

1. Power Law behaviour
2. Stable distribution condition
3. Tail index in the range (0, 2)

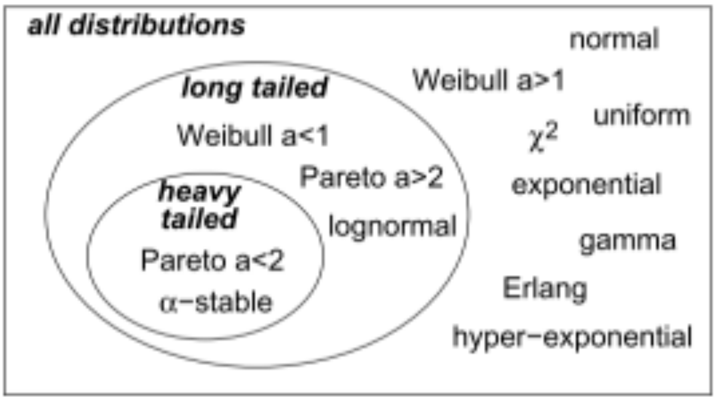


Figure 4.8: Heavy tailed distributions

Source: Feitelson, D. G. (2015). Workload modeling for computer systems performance evaluation.

Stable distributions

Heavy tailed distributions are stable distributions. If the observed latency values have a finite variance (which is not the case in heavy tailed distributions), then the distribution of average values of that dataset should follow a normal distribution. Since heavy tailed distributions have an infinite variance, above condition does not hold.

Distributions which have the same distribution as the original distribution, when aggregated are called stable distributions. We use the following aggregation function (4.16), to get the aggregated samples.

$$Xi(m) = \sum_{j=(i-1)m+1}^{im} Xj \qquad \text{...(4.16)}$$

Heavy tailed distributions have a right tail with the same tail index as the original distribution, when aggregated. Pareto distribution displays heavy tailed behaviour in the complete range it is defined, when ‘a’ is in the range (0, 2).

Tail index

In this section we focus on three different methods of calculating the tail index of a long tailed distribution. Log-log complementary graphs method can be applied to a distribution even in the absence of heavy tailed nature. Maximum likelihood and Hill estimator can be used only when the underlying latency distribution has the heavy tailed behaviour.

Log-log complementary graphs

Log-log complementary graphs (LLCD) are based on (4.10). Taking the log of both sides of equation (4.10) yields,

$$log(F(x)) = logx^{-a} = -alogx \qquad (4.17)$$

Hence plotting the log of the fraction of observations larger than x as a function of log x should lead to a straight line with slope −a, where ‘a’ is the tail index. Distributions like Pareto distribution, with tail index in the range (0, 2) results in a straight LLCD plot in the entire region it is spread. For actual latency distribution, we only observe the long tail behaviour in the final 1% of the data, when ordered in ascending order. Hence when calculating the tail index for actual workloads, we always consider only the last 1% of the dataset.

Figure 4.9 below shows the LLCD plot obtained using this method.

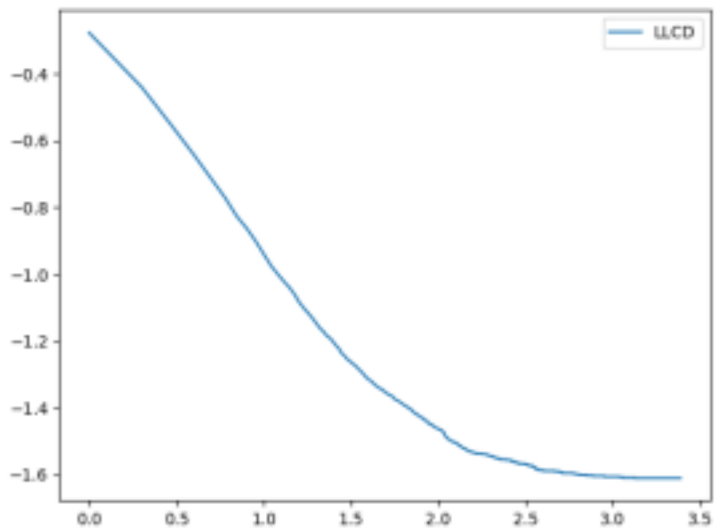


Figure 4.9: LLCD

Maximum likelihood estimation

In this method, we use standard Maximum likelihood estimation to calculate the parameters of the underlying Pareto distribution. The estimated parameter is the tail index, we are interested in. The maximum likelihood estimation of Pareto index (tail index) is given in (4.18). (‘k‘ stands for the minimum latency value)

$$a = 1/(1/n\sum_{i=1}^{n} lnxi/k) \quad \text{...(4.18)}$$

Hill estimator

Hill estimator works only when the data follows heavy tailed behaviour. It is based on Equation (4.19).

$$ak = 1/(1/k\sum_{i=1}^{k} ln(Xn - i/Xn - k)) \quad \text{.........................(4.19)}$$

X_m is the m^{th} order statistic. When only the last k samples are considered to be the tail, this is the same as maximum likelihood estimation. For different values of k, tail index is calculated and plotted. If the values converge, then it is taken as the estimate for the tail index.

When the data exhibits a power law behaviour, but not heavy tailed behaviour, this estimator does not converge.

Figure 4.10 shows the sample Hill plot.

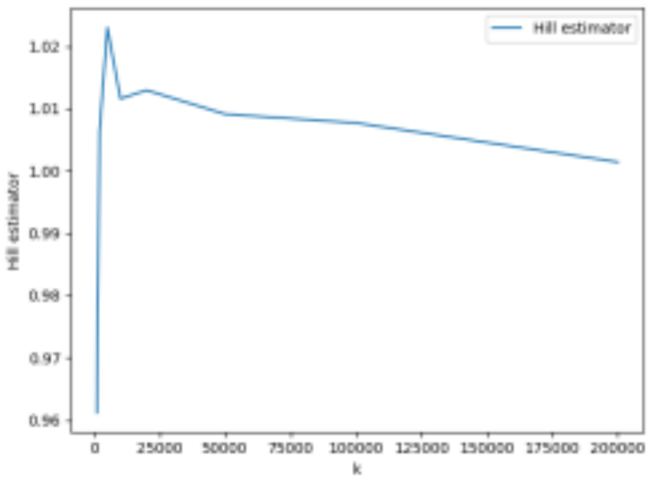


Figure 4.10: Hill plot

5. WEB SERVER ARCHITECTURES

In this section, we aim at building and analysing new high performance server architectures. We first implement the existing well known server architectures such as blocking server, non-blocking I/O (NIO) server, SEDA server. Then using actor pattern and LMAX disruptor, we extend these architectures to new architectures. We then perform an extensive analysis of all the server architectures.

5.1 Web Server Architectures

There are three main web server architectures; blocking thread per connection model, NIO model which is event driven and staged event driven architecture (SEDA). In this section we first focus on the basics of these three architectures.

5.1.1 Thread per request architecture

Thread per request model is used in RPC [56] and Java Remote Method Invocation [57]. Thread per request model is supported by modern languages and programming environments such as Java and C++.

A separate thread is allocated for each client connection. Since a thread is created for each request, synchronization operations are used to maintain correctness. The operating system transparently switches among threads. This enables to increase the CPU utilization in case where most threads are waiting for I/O operations.

To avoid the increasing number of threads, systems use thread pools. In this approach, a fixed sized (or dynamically resizing) thread pool is used. Hence, there is an upper bound of concurrently served requests. Apache [58] and IIS [59] use thread pools.

However, this approach of dropping connections affects the availability. When all the threads are running, additional requests get queued up. This causes clients to experience arbitrarily large waiting times. Figure 5.1 below depicts the class diagram for thread per request model

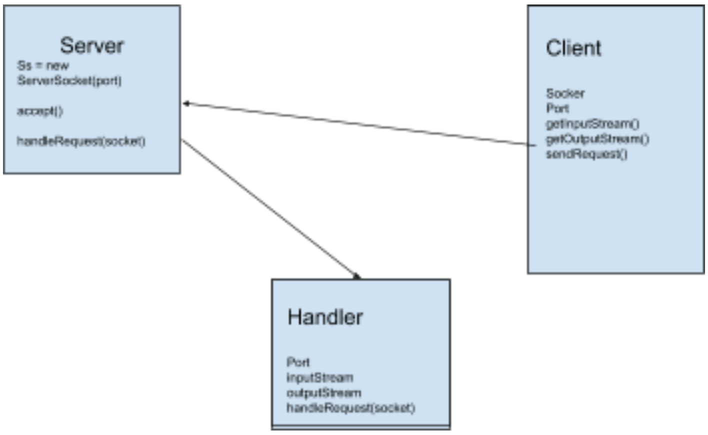


Figure 5.1: Thread Per Request Class Diagram

5.1.2 Event driven architecture

Thread per request architecture fails to scale when the workload is high. Though threads are lightweight components, context switching overhead and the stack management overhead imposed by threads is non negligible when the number of threads is high.

Event driven architecture addresses these issues by handling the I/O in a none blocking manner. It uses a single thread to handle a growing amounts of threads. In this approach, the I/O handling thread never blocks. Each connection is registered with the selector, and get its share once it is ready to perform I/O. Internally, this uses select () and epoll() system calls to check the readiness of channels. There are two variations of event driven architecture, reactor and proactor.

Reactor pattern

Figure 5.2 depicts the class diagram for reactor pattern. Reactor based NIO is the most popular approach of event driven architecture. The selector registers all the accepted sockets with it. When the channel is ready to perform the I/O, it notifies the selector. Then the selector selects this channel for I/O.

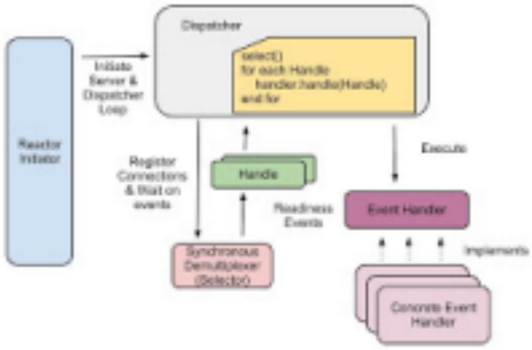


Figure 5.2: Reactor pattern

Source: https://www.javacodegeeks.com/2012/08/io-demystified.html

However, in the reactor model, there is no absolute guarantee that the event handler will do the I/O operation in a non-blocking manner.

Proactor pattern

Proactor pattern uses asynchronous I/O model. Figure 5.3 depicts class diagram for Proactor pattern.

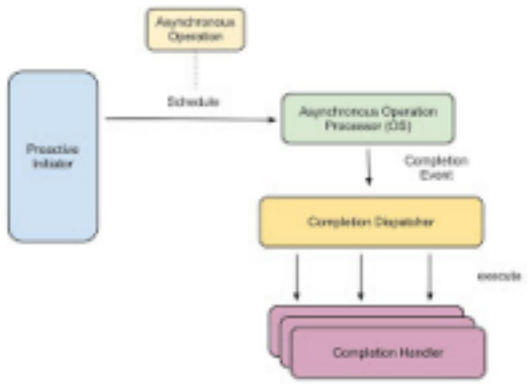


Figure 5.3: Proactor pattern

Source: https://www.javacodegeeks.com/2012/08/io-demystified.html

Proactor pattern addresses a limitation of the reactor pattern. In the reactor pattern, the selector notifies only the readiness, and does not guarantee the non-blocking execution of events. In contrast, in the Proactor pattern, the application delegates this work to the OS. Event completion handlers are triggered only when the I/O is completed.

5.1.3 Staged event driven architecture

Welsh et al. [15] have proposed a novel architecture that uses the strengths of both multi-threading and event driven notifications. The smallest unit of processing within Staged Event Driven Architecture (SEDA) is the stage. A stage consists of an input queue, output queue, a thread pool and controllers (optional).

At each iteration in the stage, a set of events are dequeued from the input queue and then processed. Number of concurrently handled requests are determined by the batching factor. Upon completing the processing of a set of events, the events are added to the output queue.

Event handlers, which contain the logic to process events, are not tightly coupled with stage operations. Unlike the original SEDA work [15], we do not employ resource controllers in the research. Figure 5.4 illustrates the structure of a SEDA-based application in the original SEDA specification [15].

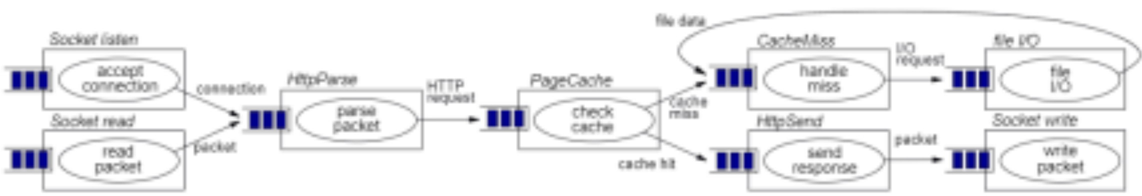


Figure 5.4: SEDA architecture

Source: SEDA: An Architecture for Well-Conditioned, Scalable Internet Services

5.2 Message Passing Architectures

There are three widely used architectures for inter thread communication; sending messages from one thread to another. In this section each of these three methods will be explored.

5.2.1 Queue

Queue is a data structure that is widely used in programming. Queue follows First-In-First-Out principle. Data items stored first will be accessed first. Queue is implemented using Arrays, Linked-lists, Pointers and Structures.

Basic Operations

1. enqueue() − store an item to the queue (added to the tail).
2. dequeue() − remove an item from the queue (using the head).
3. peek() − get the front element without removing it.
4. isfull() − checks if the queue is full.
5. isempty() − checks if the queue is empty.

Though queues have advantages such as concurrent access by many threads, increased throughput due to queuing, it has many disadvantages, as shown in [60]. Increased latency, costs of locks to maintain the correctness, write contention on the head and tail, production of more garbage objects are some of the key disadvantages of queues. Hence a more advanced method message passing; disruptor and message passing are employed.

5.2.2 Disruptor

Disruptor is a high performance message exchange mechanism [60]. Disruptor addresses the contention issues of the queue. Disruptor is based on a concept called Mechanical sympathy.

Mechanical sympathy

Mechanical sympathy accounts for how different memory allocations affect the performance. When the CPU requests data, it is searched in Register, L1, L2, L3, memory and hard disk order. Table 5.1 summarizes the typical values for each operation.

Table 5.1: Mechanical Sympathy

Latency from CPU to	CPU cycles	Time
Main memory	Multiple	~60-80 ns
L3 cache	~40-45 cycles	~15 ns
L2 cache	~10 cycles	~3 ns
L1 cache	~3-4 cycles	~1 ns
Register	1 cycle	Very quick

Following figure 5.5 depicts the structure of the disruptor.

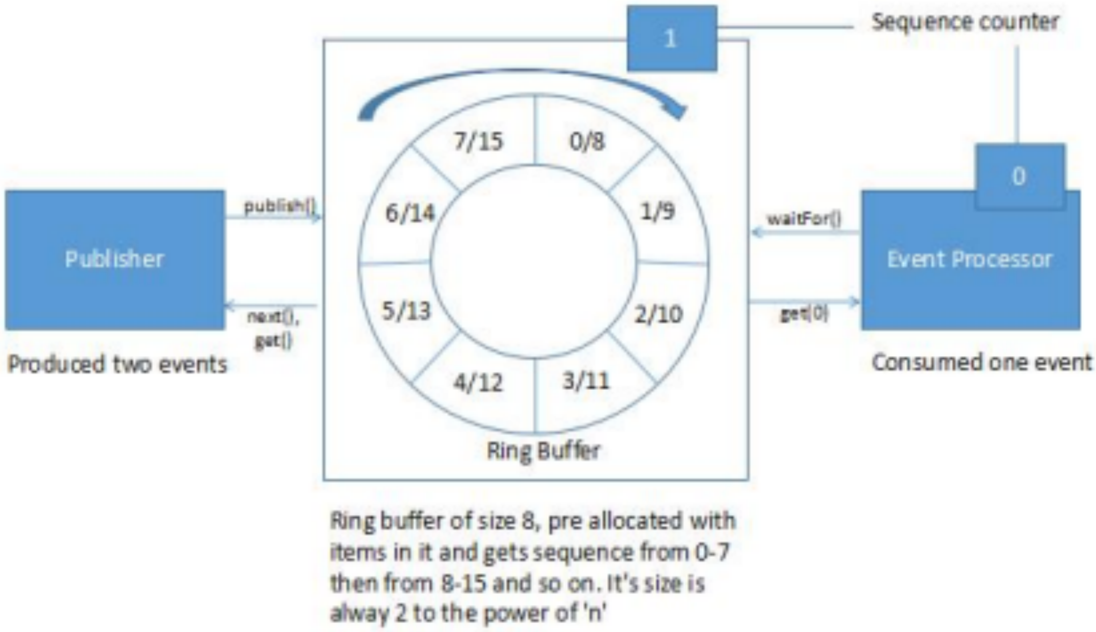


Figure 5.5: Disruptor structure

Source: https://www.baeldung.com/lmax-disruptor-concurrency

Disruptor uses a ring buffer based mechanism to pass data between two independent units of a program. Ring buffer is a pre-allocated linked list. When a producer publishes an event, all the consumers are notified. Since the buffer is pre allocated, we can safely assume that the adjacent elements of the buffer fit into the same cache line. This reduces cache miss rates.

5.2.3 Actors

An actor represents an independent computation unit (same as a thread). Unlike threads, Actors are very high level objects that communicate only using messages. Each actor has an address and a mailbox to which other actors add messages in an asynchronous manner.

There are several advantages to using Actors.

1. Can write the code without worrying about the synchronization issues
2. Supports asynchronous message passing
3. Automatic error handling

5.3 Methodology and Implementation

Using the four web server architectures, and three message passing architectures, we come up with 12 web server architectures, eight of which are novel. Table 5.2 below lists all the web server architectures we implement.

All these implementations are publicly available at [61]. Since each of the server architectures is self-explanatory, only a brief introduction to each architecture will be given.

1. Blocking: A simple blocking threaded server with a thread pool of size four

2. Blocking Disruptor: Instead of using a thread pool, a Disrupter is used to send the accepted socket to a handler. There are four handlers.

3. Blocking Actor: Threads in the original blocking server is replaced by Actors. There are four handler actors

4. NIO: Non-blocking I/O single threaded server, which is based on reactor pattern

Table 5.2: Server architectures

Name	Novelty	Multi-threading support
Blocking	Existing	Yes
Blocking Actor	Novel	Yes
Blocking Disruptor	Novel	Yes
NIO	Existing	No
NIO Disruptor	Novel	Yes

NIO Actor	Novel	Yes
NIO2	Existing	Yes
NIO2 Disruptor	Novel	Yes
NIO2 Actor	Novel	Yes
SEDA Queue	Existing	Yes
SEDA Disruptor	Novel	Yes
SEDA Actor	Novel	Yes

5. NIO Actor: The original NIO server is modified to support multi-threading. The main thread accepts, reads from the socket, and the subsequent operations are passed to the handler actor which runs in a separate thread. There are four such handlers.

6. NIO Disruptor: Same as the NIO actor model, except the actor model is replaced by a disruptor. There are four handlers.

7. NIO2: Non-blocking I/O server based on Proactor pattern

8. NIO2 Disruptor: NIO2 server is modified, and the actual processing of the request is done using an event handler. Events are passed to the handlers using a Disruptor.

9. NIO2 Actor: Same as NIO2 disruptor, except that the Disruptor and handlers are replaced with Actors.

10. SEDA Queue: Implementation of the original SEDA architecture using queues

11. SEDA Disruptor: Queues in the SEDA queue architecture are replaced with Disruptors

12. SEDA Actor: Queues and threads in the SEDA queue server is replaced with Actors.

5.3.1 Micro benchmark applications

We use the micro benchmark applications we defined in our publication [89].

5.3.2 Workload generation

For all the experiments, we use a two machine setup (connected using a LAN), where one machine hosts the server application while the other machine generates the workload. We use a separate machine to host the database.

We use Java 8, the most widely used virtual machine based language for servers to build the micro benchmarks. MySQL 5.0.27 database was used as the database application.

Synthetic workloads are used due to two main reasons, 1. Ability to change independent variables and collect data for a wide range of situations 2. Time constraints on collecting actual workloads using real systems, and our requirement to evaluate many different combinations of heap sizes, concurrency levels, and workloads.

We use apache JMeter 4.0 [51] which is widely used in workload characterization literature [63] [38]. We send the same request to the micro benchmark application, for example the same prime number is sent to the Prime service, in each user request.

We use this approach because we focus on exploring the performance only under service's peak sustainable throughput. Sending the same request reduces the impact of just in time compilation and class loading time, since the same set of Java methods are invoked in each request. Our workload generation scripts are publicly available at [64].

For each micro benchmark application, built using each 12 web server architectures, we experiment on two heap sizes (100MB and 2GB) by specifying the Xmx and Xms in Java_OPTS environment variable. For each heap size, we experiment on two different levels of concurrency, 10 and 300. Then for each concurrency level, we vary the service demand by varying the parameters in the request.

For the CPU bound micro benchmark, the service time is mainly affected by the prime number. Hence we arbitrary choose two different prime numbers, 11 and 27059 to represent low service demand and high service demand.

For the memory bound micro Benchmark, we consider two sizes, 10 and 1000 as service demands. We use integers for our calculations (four bytes per number).

We use two industry standard message sizes that are used in Middleware performance testing [65] as our service demands for network I/O bound micro benchmark, 10B, and 1KB. We do not alter the service demands for the database I/O bound micro benchmark.

In total, we collect data for 362 number of combinations. We run our experiments for a period of 15 minutes for each combination of web server architecture, micro benchmark, heap size, concurrency level, and service demand. The total dataset size is 828GB. Due to space limitations, we have not published this online, yet can be made available on request.

In order to remove Java just-in-time compilation and class loading effects from our results, we remove the first M minutes results using JMeter Splitter [66]. We observe an almost constant throughput, after five minutes of test initiation. Hence we chose M to be five minutes. We collect Java garbage collection (GC) logs and load average statistics using SAR [67] reports and hardware counters using perf [68].

5.4 Experiment Setup

We use a bare metal setup for our web server architecture performance tests. For each machine (client, server and database host) we use a server-class machine (Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz, 8 GB of RAM, 1TB hard disk) connected using Gigabit Ethernet.

5.5 Results

In our tests, we record the configuration (heap size and etc), latency, throughput and the values extracted from garbage collection logs, SAR reports and perf tests. In total we collect 101 number of features for each configuration. Due to space limitations we will not present the results table here. The complete result sheet is published publicly in [70] and in Appendix - A.

5.6 Discussion

In this section, we use the following terminology to denote specific configurations.

1. Low heap = 100MB
2. High heap = 2GB
3. Low concurrency = 10
4. High concurrency = 300
5. Low service demand
    a. I/O = 10B
    b. CPU = isPrime(11)
    c. Memory = merge-sort(10)
6. High Service demand

a. I/O = 1KB
b. CPU = isPrime(27059)
c. Memory = merge-sort(1000)

5.6.1 Blocking architectures

IO bound micro benchmark

We observe that Blocking architecture gives significant throughput compared to Blocking Actor and Blocking Disruptor architectures, for the following configurations.

1. Low heap, low concurrency and low service demand
2. Low heap, high concurrency and high service demand
3. High heap, high concurrency and high service demand

For example, we observe a throughput of 7565 requests per second for Blocking architecture for the low heap, low concurrency and low service demand configuration, where the respective throughput values of Blocking Actor and Blocking Disruptor are 7473 and 5766 requests per second. We explain this behaviour as follows.

For each three configurations we mentioned above, we observe that the average garbage collection pause, number of CPU cycles and number of executed instructions are very low in Blocking architecture. Since blocking architecture is the minimal overhead implementation, compared to other two Blocking architectures, it uses less memory and instructions. This causes high throughput for the Blocking Server.

We also observe that Blocking Disruptor architecture performs poorly for the following two configurations.

1. Low heap, low concurrency and low service demand
2. High heap, low concurrency and low service demand

For example, we observe that for the low heap, low concurrency and low service demand configuration, Blocking Disruptor throughput equals to 4675 requests per second, whereas for the Blocking and Blocking Actor architectures the respective throughput values are 5876 and 5743 requests per second. We explain this behaviour as follows.

Blocking Disruptor architecture consumes more memory compared to the other two architectures. Hence we observe very high full garbage collection pauses for the Blocking Disruptor architecture. Garbage collection events are stop the world events which halt the application threads. Hence the performance suffers.

We also observe very low throughput in Blocking Actor architecture, compared to the other two architectures, in the following configurations.

1. Low heap, high concurrency, low Service demand
2. High heap, high concurrency and low service demand

For example for the low heap, high concurrency, low service demand configuration, we observe a throughput of 2567 requests per second for the Blocking Actor architecture, whereas the respective values for the Blocking and Blocking Disruptor are 7008 and 5907 requests per second. We explain this behaviour as follows.

Blocking Actor shows a high idle processing time, which is also reflected in load average statistics. This indicates that Blocking Actor is not able to fully utilize its resources. We believe having only four actors as workers is the main reason for this behaviour. If the number of actors are increased to a higher value, the throughput can be increased.

CPU bound micro benchmark

We observe that with low heap size, low concurrency and low service demand, Blocking Disruptor performs very poor compared to other two Blocking architectures. For example, for this configuration we observe a throughput of 5732 requests per second for Blocking Disruptor, whereas for the other two architectures, the throughput values are greater than 7400 requests per second.

Analysis of the hardware and software counters revealed high full garbage collection pauses as the main reason for getting this performance degradation. Also, we observe high task clock rates, high context switches, high CPU migrations and high cache misses in the Blocking Disruptor architecture.

Although Disruptor is designed with low cache misses in mind, high garbage collection operations make its value a little. When garbage collection happens, the application threads are halted. This causes high context switches, which eventually leads to high CPU migrations. High CPU migrations causes' high cache misses, since the thread changes the processor on which it runs.

We also observe that with low heap, low concurrency and high service demand, our novel Blocking Actor architecture performs significantly better than the other two architectures. We observe a throughput of 7228 requests per second in the Blocking architecture, whereas the maximum observed throughput for the other two architectures is 6704 requests per second. This behaviour is also seen in CPU hardware performance counters, idle time percentage and load average. We observe a low idle time percentage (hence more useful work is done in application), and high load average (CPU is fully utilized).

Memory bound micro benchmark

We observe that with low heap, low concurrency, low service demand, the performance of Blocking Disruptor architecture is significantly low. For example, for the above scenario, the throughput of Blocking Disruptor is 5741 requests per second, whereas the minimum throughput of other two architectures is 7500 requests per second. We explain this behaviour as follows.

For this configuration, Blocking Disruptor shows significantly high context switches, a very high CPU migration number and a high number of cache misses. This result is non-trivial because the Disruptor was originally proposed with low cache misses in mind. But our results suggest that Disruptor is not a silver bullet, and for some cases, adding a Disruptor makes the system perform poorer.

We also observe that for low heap, high concurrency and low service demand configuration, the Blocking Actor performance is significantly low. For example, the throughput of Blocking Actor is 2687 requests per second, whereas the minimum throughput of other two Blocking architectures is 6689 requests per second. We explain this behaviour as follows.

Idle time percentage is very high in Blocking Actor implementation for the above configuration. Low load average values observed for this configuration also supplements this factor. This indicates that the CPU is not fully utilized. Hence, more handler Actors should be added to increase the throughput.

DB bound micro benchmark

We do not observe a significant performance difference between the three Blocking architectures, for the DB bound micro benchmark. For all configurations we observe a throughput close to 1200 requests per second. This behaviour can be described as follows.

In this DB bound micro benchmark, an external I/O operation is performed (the Database access). This adds a high latency. Hence the overall system is bound by the speed of Database access and network speed.

5.6.2 NIO architectures

We observe that for all configurations of I/O bound, CPU bound and memory bound micro benchmarks, NIO architecture performs significantly better than the other two architectures. For example, for the low heap, low concurrency and low service demand configuration of I/O bound micro benchmark, we observe a throughput of 3538 requests per second for the NIO architecture, whereas the maximum of other two architectures is 1334 requests per second. This behaviour can be explained as follows.

When the actual processing of the request is very low, the overhead of transferring the processing to another worker is higher than that of doing it in the main thread itself. This leads to decreased throughput in multi-threaded implementations.

We also observe a very low number of context switches and very low number of CPU migrations in the NIO architecture. Since the NIO architecture uses only a single thread, this result can be accepted. Since the other two NIO architectures use multiple threads, they incur higher number context switches and higher number of CPU migrations.

In contrast, we observe a drastic performance gain in our newly proposed NIO Actor and NIO Disruptor architectures, for the Database I/O bound benchmark. For example, in the DB bound micro benchmark, for low heap, low concurrency configuration, we observe throughputs of 635 and 653 requests per seconds for NIO disruptor and NIO Actor architectures respectively, whereas the respective value for the NIO architecture is 333.46 requests per second.

Analysis of the CPU counters revealed high I/O wait percentage and the high idle time percentage as the main reasons for the above observation. DB calls require a significant amount of time. In the NIO architecture, the main and the single available thread halts until the database response is available. This negatively affects the performance. In the new architectures, NIO Actor and NIO Disruptor, this heavy waiting is done using another thread. This leads to increased throughput in the new NIO architectures.

5.6.3 NIO2 architectures

We observe that for each micro benchmark, for each heap size, for each level of concurrency and service demand, NIO2 architecture outperforms NIO2 Actor and NIO2 Disruptor by a significant margin. For example, in the DB bound micro benchmark, for low heap, low concurrency, we observe a throughput of 1310 requests per second for NIO2, whereas the maximum throughput of other two architectures is 287 requests per second. This result is a bit surprising, and we reason this behaviour as follows.

NIO2 is inherently multi-threaded; each request is handled by three different threads. Also, it can deploy many numbers of threads to support a given concurrency level. In the NIO2 Actor and NIO2 Disruptor architectures, we hand over the processing to an external thread. Yet, we have fixed the number of handlers to four. Hence, the overall operations are constrained by the number of handlers. This drastically drops the throughput.

We also observe that for all the configurations, the task clock is very low in the NIO2 architecture. This indicates that NIO2 has performed less work compared to NIO2 Actor and NIO2 Disruptor. Yet, as we have already shown above, throughput is maximum for NIO2 compared to NIO2 Actor and NIO2 Disruptor. We reason this behaviour as follows.

NIO2 Actor and NIO2 Disruptor architectures employ additional processing for a request, due to the addition of handlers. This only includes more processing for a request, and does not help increase throughput.

5.6.4 SEDA architectures

We observe a significantly low throughput for the SEDA Disruptor architecture, for each benchmark, each heap size, for each concurrency and for each service demand. We explain this behaviour as follows.

SEDA disruptor architecture displays significantly high garbage collection pauses, very high page faults. This suggests that Disruptor architecture consumes more memory than others. The garbage collection delays and time consumed for page faults impact the throughput.

We also observe significant throughput gains in our novel SEDA Actor architecture, for the following scenarios.

1. I/O bound micro benchmark for all heap sizes, all concurrency levels and high workloads.
2. CPU bound micro benchmark for high heap size, all concurrency levels and all workloads.
3. Memory bound micro benchmark for high heap, high concurrency and high service demand.

We explain this behaviour as follows.

SEDA Actor architecture incurs very less garbage collection overheads, as resembled in the accumulated garbage collection pause times. Also, SEDA Actor shows a significantly low number of context switches, page faults. These factors improve the throughput.

5.7 Summary

In this section, we first described the 12 web server architectures, eight of which are novel. We then presented a micro benchmark application as a tool to isolate different types of service calls. We then performed an extensive performance analysis of each web server architecture for different number of concurrent users, heap and service demands. Our analysis shows that the novel proposed server architectures outperform the existing architectures, and provides insights into further improvements.

6. JAVA MICROSERVICES TAIL LATENCY ANALYSIS

All the contributions on this topic appear in our publication, Tennage et al. [89].

7. SCALABILITY

7.1 Introduction

Scalability of a system can be measured in two ways.

1. Hardware Scalability
2. Software Scalability

Hardware scalability refers to how the system scales when more hardware resources are added. For example, if a certain server gives a throughput of x, what will be the throughput when the number of nodes are doubled?

In the software scalability, given a fixed hardware configuration, we find the scalability characteristics of the application under different concurrency levels.

Consider a simple web application running on a machine with fixed hardware. When the application is run with 100 concurrent users, assume a 1000 transactions per second maximum throughput. When the concurrency level is increased to 200, ideally a throughput of 2000 transactions per second should be observed. However, the maximum throughput at a 200 concurrency level is less than 2000 transactions per second. When the level of concurrency is further increased, the throughput starts to display retrograde behaviour.

Exhaustive capacity planning can explore this issue. In capacity planning, the throughput of the system is measured while increasing the level of concurrency until the concurrency level which shows retrograde throughput behaviour is found. Yet, exhaustive capacity planning requires a larger budget and a substantial amount of time. Instead Universal Scalability Law (USL) proposes an analytical method which is effective in time and budget.

In this section, first the USL for software is presented by extending Amdahl's law. Then the USL is applied for a class of server workloads called middleware.

7.2 Amdahl's Law for Software Scalability

Amdahl's Law calculates the reduction of speed up due to the part of the program that runs sequentially [85]. This can be represented using (7.1).

$$Speedup = 1/((1-Fraction-enhanced)+(Fraction-enhanced/speedup-enhanced))$$

....(7.1)

In this equation, the $Fraction_{enhanced}$ is the portion of runtime to reduce. $Speedup_{enhanced}$ is the inverse of the fractional time reduction.

If fraction enhanced is denoted by π and speed up enhanced (fractional time reduction) by φ, equation (7.1) can be written as (7.2).

$$Ssw = 1/((1-\pi) + \phi\pi) \quad \text{..........(7.2)}$$

Let σ = 1 - π, where σ is the serial fraction of the workload.

Assume π (fraction enhanced) can be divided into N parts. Then φ = 1/N.

Then equation (7.2) is reduced to (7.3) and (7.4)

$$Ssw = 1/(\sigma + 1/N(1-\sigma)) \quad \text{..........(7.3)}$$

$$Ssw = N/(1 + \sigma(N-1)) \quad \text{..........(7.4)}$$

Equation (7.4) is the equation of Amdahl's Law for software. Using this formula, the USL equation for software scalability is derived by adding the impact of interprocess communication among different users.

7.3 Universal Scalability Law for Software Scalability

Gunther et al. [86] have provided a formal equation for software scalability as in (7.5). When there are N number of concurrent users, there will be a maximum of N(N-1) number of interactions among user processes. To capture this behaviour, a new parameter β, which is called coherency is added to (7.4).

$$Csw(N) = N/(1 + \alpha(N - 1) + \beta N(N - 1)) \quad \text{................(7.5)}$$

Being a rational function, equation (7.5) can be differentiated with respect to N. The value of N at which Csw(N) is maximum is shown in (7.6). Then the maximum value of Csw(N) is Csw(N*)

$$N1/2 = \sqrt{(1 - \alpha)/\beta} \quad \text{..(7.6)}$$

In software scalability tests, scalability is measured as a function of the number of users N. It is assumed that the underlying hardware platform is fixed for all measured points of N.

To summarize, USL is a rational function of three parameters

1. Level of concurrency (N)
2. Contention (α) which is the serial fraction of the workload
3. Coherency (β) which is the penalty for interprocess communication.

In the following section, USL for software is calculated for a class of server workloads; middleware. Widely used Enterprise Integrator WSO2 EI is used for this purpose.

7.4 WSO2 Enterprise Integrator Dataset

The WSO2 EI is an open source product distributed under the Apache Software License v2.0. WSO2 EI allows message routing, mediation, transformation, logging, task scheduling, failover routing, load balancing, and more. In this section a basic use case of EI, which is Direct Proxy or Simple Pass-Through Proxy is used.

A simple Netty Echo service is deployed as the backend for the WSO2 EI. Three JMeter instances are deployed in order to handle a large concurrency level, and to ensure that JMeter nodes do not run out of resources when running in very high concurrency levels (usually more than 1000). Figure 7.1 below illustrates the EI setup.

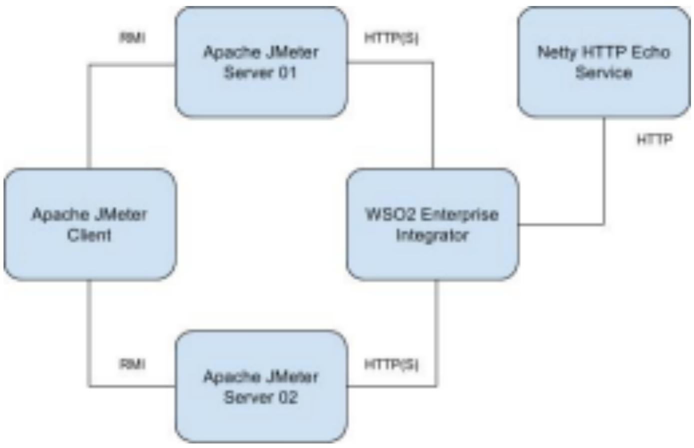


Figure 7.1: EI setup

Source: https://github.com/ThishaniLucas/performance-ei/tree/perf-test

7.5 Experimental Setup

WSO2 EI 6.4.0 is used for the experiments. Three different industry standard message sizes; 500B, 1KB, and 10KB are tested. For each message size, four different concurrency levels, 100, 200, 500, and 1000 tested. For each message size and concurrency level configuration, tests are run in c5.xlarge Amazon EC2 instance for 15 minutes. The maximum heap size is fixed to 4GB and backend service delay to zero seconds. The first five minutes results from the JTL files are removed in order to get only the steady state results.

7.6 Results and Discussion

Table 7.1 summarizes the performance results.

USL package in the R language is used to compute the universal scalability law parameters. Table 7.2 summarizes the USL parameters for this dataset. Figure 7.2 depicts the USL curves for three message sizes.

As the message size increases, we observe a significant drop in throughput. Using USL parameters, we can identify that increasing the message size increases contention and coherency parameters. For example, when the message size increases from 500B to 10KB, contention (α) significantly increases from 7.306e-02 to 7.494e-02 and the coherency (β) increases from 6.554e-07 to 9.883e-06. As a result, the concurrency level which starts to display the retrograde behaviour decreases from 1189 at 500B message size to 306 at 10KB message size.

Table 7.1: Universal law of scalability performance results

Message Size (KB)	Concurrency (N)	Throughput (requests per second)	Average Latency (ms)
500B	100	17588.2	5.63
	200	19509.07	10.17
	500	19940.62	24.92

	1000	19764.01	50.5
1KB	100	16667.76	5.93
	200	18175.66	10.87
	500	18235.69	27.23
	1000	18255.44	54.69
10KB	100	13173.19	7.5
	200	13937.52	14.22
	500	13434.7	37.1
	1000	13203.39	75.63

Table 7.2: USL parameters

Message Size	α	β	Max users (N*)	Max throughput (requests/second)
500B	7.306e-02	6.554e-07	1189	19921.33
1KB	7.372e-02	2.413e-06	620	18343.21
10KB	7.494e-02	9.883e-06	306	13852.88

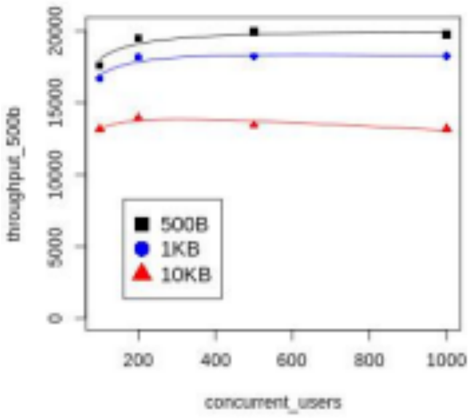


Figure 7.2: USL curves

Contention (α) and cohesion (β), reveal the factors that hinder the performance of a software system. If a high α value is observed, then the software should be modified to minimize serialization. If a high β is observed, it reflects that the software system has many inter thread communication. Hence, inter thread communication should be minimized.

7.7 Summary

When developing an application software, it is important to focus on the scalability characteristics. USL provides a more analytical approach to explore this problem. This section explored USL for software and a use case. First the USL equation was derived by extending Amdahl's Law. Then using R language library usl, the parameters for the WSO2 EI simple proxy was found. Finally, the scalability characteristics of EI were discussed using USL.

8. DISCRETE EVENT SIMULATION

8.1 Introduction

As we have already shown in the above sections, performance testing of web servers requires a large time and cost. In practice, sometimes it is not required to get the exact performance numbers (for example the exact latency), but general trends about the performance is sufficient. Discrete event simulation (DES) can be used for such scenarios.

DES models a system as a discrete sequence of events in time. An event in this context is an item that changes the state of the system. DES is used in diagnosing process issues, modelling hospital applications, and etc.

Since we employed the closed system model throughout this research, we will use the closed system model for the following DES experiments.

In this section, we first discuss the basic concepts of DES. Then, we explore the DES package-Simpy. The code for a single server closed loop performance test is presented afterwards. We then extend the simple version of the single server to multiple servers, which has inter-service calls (abstraction for microservices).

8.2 Definitions

Discrete event simulation (DES) simulates the behaviour of a process. In DES, a system is modelled as a series of events that occur over time.

There are three major DES paradigms: activity oriented, event oriented, and process oriented.

Activity Oriented Paradigm

In Activity Oriented Paradigm, time is broken into small increments. At each time point, the code would look around all the activities.

Event Oriented Paradigm

In Event Oriented Paradigm, the time counter is advanced to the time of the next event. This approach saves the CPU cycles.

Process Oriented Paradigm

In Process Oriented Paradigm activity is modelled as a process. This is the widely used approach in current state-of-the art DES systems.

The DES framework SimPy, uses a process oriented paradigm.

8.3 SimPy

SimPy is a process-based discrete-event simulation framework. Processes in SimPy are implemented using generator functions. Processes are used to model the web servers and Clients. SimPy also has shared resources, for example SimPy resources.

8.3.1 Major concepts

Yield

A SimPy process can be yielded. When a process is yielded, the execution returns from the process for the given event, and returns. The process resumes upon the completion of the event.

Timeout

Timeout is an event that gets executed after a timeout.

Process interactions

There are two main process interactions in SymPy:

a. Waiting for another process to finish
b. Interrupting another process.

Shared resources

Shared resources can be shared among other different resources (for example the queue between the clients and the server is a shared resource called a Pipe)

8.4 Closed System DES Simulation

The setup depicted in Figure 8.1 is used as a model for the initial closed model DES setup. The workload generator represents a set of clients. Figure 8.2 shows the DES abstraction for the model shown in Figure 8.1.

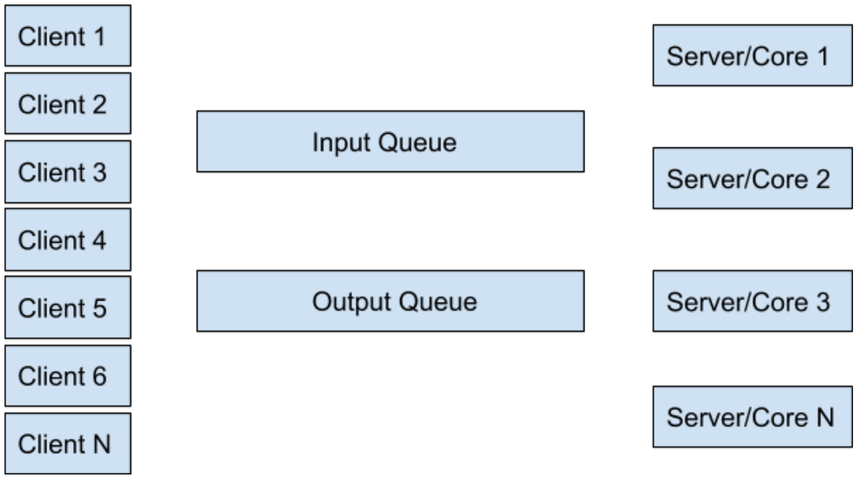


Figure 8.1: DES abstraction

In this setup, N clients are used. The server application runs in an N core machine. It is assumed that the server application can handle N number of requests concurrently. Two queues are used: input queue and output queue. Each client adds a requests to the input queue. The requests gets queued in the queue and each core fetches the request at the top and processes them. Upon completion, the request is added to the output queue. Then the response is received by the client. Upon receiving the response, each client sends the subsequent request.

```
1 import random
2 import simpy
3
4 SEED = 42
5 average_processing_time = 0.025
6
7 response_times =[]
8 queue_lengths = []
9 waiting_times = []
10
11 concurrency = 1000
12 num_cores = 4
13
14
15 def client(env, out_pipe, in_pipe, i):
16     global response_times
17     while True:
18         processing_time = random.expovariate(1/average_processing_time)
19         arrival_time = env.now
20         d = {1: processing_time, 2: i, 3: arrival_time}
21         out_pipe.put(d)
22         response = yield in_pipe.get(filter=lambda x: True if x[2] == i else False)
23         response_time = env.now - arrival_time
24         response_times.append(response_time)
25
26
27 def server(env, in_pipe, outpipe):
28     global queue_lengths
29     global  waiting_times
30     while True:
31         request = yield in_pipe.get()
32         processing_time = request[1]
33         arrival_time = request[3]
34         waiting_time = env.now - arrival_time
35         waiting_times.append(waiting_time)
36         queue_length = len(in_pipe.items)
37         queue_lengths.append(queue_length)
38         yield env.timeout(processing_time)
39         outpipe.put(request)
40
41
42 random.seed(SEED)
43
44 environment = simpy.Environment()
45 in_pipe=simpy.Store(environment)
46 out_pipe=simpy.FilterStore(environment)
47
48 for i in range(concurrency):
49     environment.process(client(environment, in_pipe, out_pipe, i))
50
51 for i in range(num_cores):
52     environment.process(server(environment,in_pipe, out_pipe))
53
54 environment.run(1000)
55
56 response_times=[x*1000 for x in response_times]
57 waiting_times=[x*1000 for x in waiting_times]
```

Figure 8.2: Single server pseudo code

The number of clients (concurrency) are specified in line 11, and the number of cores at line 12. This program has two process methods; client and server.

8.4.1 Client process

Line 15-24 shows the client process. Parameter env is the Simpy environment in which the process runs. in_pipe is the input queue to the client (to which the server puts the responses). out_pipe is the output pipe, into which the client puts the requests. i is the identity of the client.

First, the client process generates a pseudo random number using exponential distribution, using the given processing rate. Then it keeps track of the arrival time. Then the request is put to the out_pipe.

The client process waits until it gets the response to its request. All the responses for each client request is put to the shared in_pipe. A special pipe of type FilterStore is used as the in_pipe. Using the lambda function, the relevant response is received by the client.

8.4.2 Server process

In a continuous loop, the server checks for new requests. Once the server receives a request from the in_pipe, it extracts it. Then the server calculates the time difference between starting the processing of the request and the request creation time. Then the server core yields the corresponding processing time.

This code is run for three different concurrency levels (100, 200, and 500) and for each concurrency level, the number of cores (1, 2, and 4) are varied. Table 8.1 below shows the results.

Table 8.1: Closed System DES Results

Concurrency	Number of Cores	Average Latency (Time Steps)	99 Percentile Latency (Time Steps)	Throughput (Request Per Time Step)
100	1	2496.148	3151.9094	39.35
100	2	1252.6918	1566.3316	80.25
100	4	624.48957	792.1198	159.88
200	1	4985.5102	5917.101204	39.35
200	2	2503.9160	2950.10	80.25
200	4	1248.598	1473.52731	159.88
500	1	12415.056	14242.556	39.35
500	2	6248.57563	6968.6093200	80.25
500	4	3118.48955	3471.48797	159.88

8.4.3 Results and discussion

We observe that when the level of concurrency increases, the average latency and 99 percentile latency increase. For example, when the core count is fixed at four, when concurrency increases from 100 to 200, the average latency increases from 624 time steps to 1248 time steps. Also, the 99 percentile latency, increases from 792 time steps to 1473 time steps. This behaviour is explained as follows.

When the concurrency increases, the waiting time increases. This leads to an increased latency values.

Second, we observe that the level of concurrency does not impact the throughput. When the level of concurrency is varied from 100 to 200 and 500, the throughput remains constant at 39. This behaviour validates the theoretical proof; in a closed system, throughput is independent of the level of concurrency, and depends only on the service rate [87].

Third, it is observed that when the number of cores increases, average latency and 99 percentile latency decreases. For example, when the concurrency is fixed at 500, when the number of cores is increased from one to two, the average latency reduces from 12415 time steps to 6248 time steps, whereas, the 99 percentile latency reduces from 14242 to 6968 time steps. This observation can be explained as follows.

When the number of cores increases, the requests which are queued in the server input queue, get scheduled faster; thus reducing the queue waiting times. Hence the response time decreases.

Finally, we observe that when the core count increases, the throughput increases. For example, when the number of cores is increased from one to four, the throughput increases from 39.5 to 159.8. When the number of cores is increased, the amount of work that are done in a given time period increases. Hence, the throughput increases.

8.5 Modelling Interservice Calls

In this section, we extend the above closed system simulation to support interservice calls. Figure 8.3 depicts the DES abstraction for a system with one interservice call.

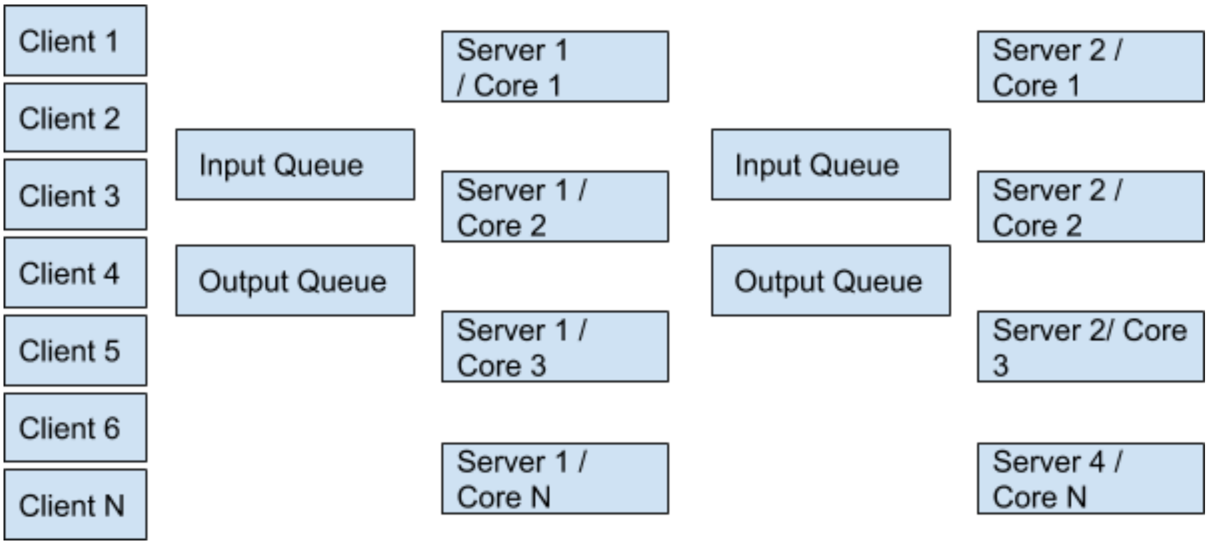


Figure 8.3: Interservice Calls DES abstraction

In this abstraction, two servers are used. The clients send requests into the input queue of the first server. Upon the completion of processing, server 1 puts the partially processed request to the input queue of server 2. Server 2 then processes for completion and puts the response to server 2's output queue. Server 1 forwards the response back to the client.

Figure 8.4 shows the pseudo code for interservice calls.

```
import random
import simpy
SEED = 42
average_processing_time = 0.25
response_times =[]
queue_lengths = []
waiting_times = []
concurrency = 100
num_cores = 4

def client(env, out_pipe, in_pipe, i):
    global response_times
    while True:
        processing_time_1 = random.expovariate(1 / average_processing_time)
        processing_time_2 = random.expovariate(1 / average_processing_time)
        arrival_time = env.now
        d = {1: processing_time_1, 2: processing_time_2, 3: i}
        out_pipe.put(d)
        response = yield in_pipe.get(filter=lambda x: True if x[3] == i else False)
        response_time = env.now - arrival_time
        response_times.append(response_time)

def server_1_1(env, in_pipe, out_pipe):
    while True:
        request = yield in_pipe.get()
        processing_time = request[1]
        yield env.timeout(processing_time)
        out_pipe.put(request)

def server_1_2(env, in_pipe, out_pipe):
    while True:
        request = yield in_pipe.get()
        out_pipe.put(request)

def server_2(env, in_pipe, out_pipe):
    while True:
        request = yield in_pipe.get()
        processing_time = request[2]
        yield env.timeout(processing_time)
        out_pipe.put(request)

random.seed(SEED)

environment = simpy.Environment()
in_pipe_1=simpy.Store(environment)
in_pipe_2 = simpy.Store(environment)
in_pipe_3 = simpy.Store(environment)
out_pipe = simpy.FilterStore(environment)

for i in range(concurrency):
    environment.process(client(environment, in_pipe_1, out_pipe, i))

for i in range(int(num_cores/2)):
    environment.process(server_1_1(environment, in_pipe_1, in_pipe_2))
    environment.process(server_1_2(environment, in_pipe_3, out_pipe))

for i in range(num_cores):
    environment.process(server_2(environment, in_pipe_2, in_pipe_3))

environment.run(1000)
```

Figure 8.4: Interservice calls, Pseudo code

Two pseudo random numbers are generated, one for server 1 processing time and the other for server 2 processing time. Server 1 process is divided into two methods, server_1_1 (for actual processing) and server_1_2 (for response forwarding).

8.5.1 Results and analysis

Table 8.2 below shows the results.

Table 8.2: Interservice calls DES results

Concurrency	Number of Interservice Calls	Average Latency (Time Steps)	Throughput (Requests per Time Step)	99 Percentile Latency (Time Steps)
100	0	6238.893	159.88	7992.514
100	1	12539.17644	79.3	16485.4686
100	2	12653.462	78.54	15825.8872
200	0	12442.118	159.88	14902.873
200	1	24913.124596	79.3	31621.465
200	2	25154.7919	78.54	29512.536
500	0	30830.13843	159.88	35722.7764
500	1	61063.3348	79.3	70531.2472
500	2	61774.7721	78.54	70094.39

We first observe that when the number of inter service calls increases, the average latency and 99 percentile latency increase. For example, when the number of interservice calls increases from zero to one, the average latency increases from 6238 to 12539 time steps, when the concurrency is fixed at 100. This behaviour can be explained as follows.

When the number of interservice calls increases, each request has to stay at increasing number of queues. This increases the accumulated queue waiting time for a given request. This waiting time causes the average latency and 99 percentile latency to increase.

Second, we observe that when the number of interservice service calls increases, the throughput decreases. For example, when the number of interservice calls increases from zero to one, the throughput decreases from 159.88 to 79.3. This observation can be explained as follows.

When the number of interservice calls increases, the queue waiting times increases significantly. This leads to an increased round trip time response times, thus reducing throughput.

8.6 Summary

In this section, we made the following contributions.

1. Implementing closed system model performance testing using DES.
2. Show that when concurrency increases the response time increases.
3. Show that throughput is independent of concurrency in a closed system model.
4. Show that when the number of cores increases, the latency decreases
5. Show that when the number of cores increases, throughput increases
6. Show that when number of interservice calls increases, the latency increases
7. Show that when the number of interservice calls increases the throughput decreases.

9. LOAD BALANCING

9.1 Introduction

Load balancing distributes incoming traffic across a set of backend servers or shapes them. With the advent of API management and service meshes, load balancers are becoming an essential part of most architectures.

In general, it is believed that load balancing reduces latency and improves throughput. However, this view ignores the overhead introduced by the load balancer. A load balancer does not always improve performance, and, in some cases, load balancing can degrade performance.

This section explores the impact of load balancing. The following are the major findings.

- With a backend service with low CPU-bound use cases, single-server performance is better than two servers and three servers with a load balancer, in both average latency and throughput.
- When the backend service's CPU usage is moderately high, a load balancer with two and three server setups exhibit performance gains.
- We only observe a linear speedup with the number of servers only when CPU usage is very high.
- There is no difference in 99 percentile latency values between three-server and two-server configurations; however, there is a significant variation between one-server and two-server configurations.

In conclusion, we argue that the overhead introduced by a load balancer should be carefully considered in capacity planning.

9.2 Definition

Distributing traffic across a set of servers is known as load balancing. Modern web systems receive very high traffic that makes it impossible to serve them using a single-server instance. Service providers use a load balancer to distribute traffic across multiple replicas and provide high availability. A load balancer provides the following functionalities.

1. Acts as a gateway for all the requests (i.e., a single entry point)
2. Routes traffic across a set of servers
3. Helps achieve service level objectives by reducing latency and increasing throughput by scaling the system.
4. Improves utilization of each backend server by optimally distributing traffic
5. Avoids backend servers going beyond peak utilization.
6. Provides failure tolerance by automatically identifying failed backend servers
7. Supports automatic scaling of backend servers

Over the years, we have observed several cases where adding a load balancer and a set of replicas slowed down the system. However, this only happens in some use cases. We designed and carried out an experiment designed to confirm this observation and to pin down the conditions under which load balancing slows down the system.

We first describe our experimental setup, high-level architecture, workload generation using JMeter, load balancing application, and back-end web service. Then, we provide a detailed discussion of our observations.

9.3 Experiment Setup

The setup includes clients, load balancers, and backend servers. Backend servers are the servers to which we want to load balance the requests. We conducted the experiments on three configurations as shown in Figure 9.1, 9.2 and 9.3. In the first setup (Figure 9.1), we did not use a load balancer. In setups 2 and 3 (Figures 9.2 and 9.3), we used a load balancer and distributed the incoming traffic from the client among two and three backend servers, respectively.

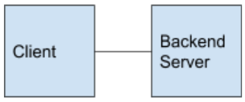


Figure 9.1: Single service

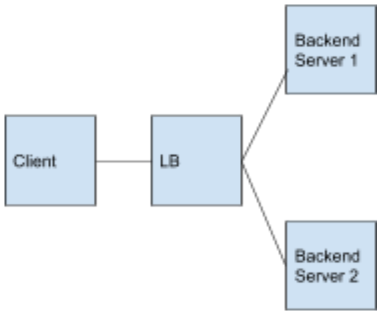


Figure 9.2: Two services

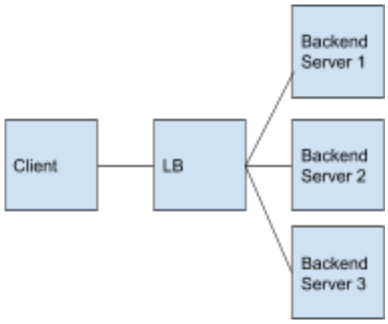


Figure 9.3: Three services

We used JMeter, a widely used load testing tool, to simulate the virtual users. At a given concurrency level (500 in our tests), JMeter sends requests to the configured endpoint (address of the services for single services configuration, the address of the load balancer for two services and three services configurations). We verified that JMeter has enough hardware resources to handle the given concurrency level.

We used NGINX [88] as the load balancer for two-service and three-service configurations. We utilized a round-robin load balancing algorithm, since each request for a given scenario has the same computational complexity.

As the backend service, we used a service that adapts a well-known CPU benchmark, first introduced by SysBench manual. This application tests whether the given number in the request is prime or not and returns a true/false response to the client. We used Java as the implementation language and Spring Boot as the framework, owing to their wide adoption.

In our prime testing web service, we checked for four number of prime numbers, 11, 541, 66601 and 1303031 that represent different CPU workloads (the prime checking application's computational complexity is proportional to the prime number provided). Hence, the CPU intensity increases with an increasing prime number (CPU-Intensity(11) < CPU-Intensity(541) < CPU-Intensity(66601) < CPU-Intensity(1303031)). These four numbers represent different levels of CPU utilization of the application.

To make our results reproducible, we ran all our tests in Amazon EC2. Table 9.1 below summarizes the hardware configurations we used.

Table 9.1: Hardware configurations

Machine	Instance Name	Number of virtual CPUs	Memory (GB)
JMeter	m4.2xlarge	8	32
Backend services (One per service)	m4.xlarge	4	16
Load Balancer	m4.xlarge	4	16

9.4 Results and Discussion

Table 9.2 below summarizes the results for each configuration. In the following discussion, we use the notation isPrime(x) to denote the set of requests that have x as the prime number. For example, isPrime(11) denotes the test where we send the number 11 in the request to check whether 11 is prime.

Table 9.2: Load balancing Results

Number of Backend services	Prime Number	Average Latency (ms)	Throughput (Requests Per Second)	99 percentile Latency (ms)
Single service	11	28	19364	99
Single service	541	28	17804	105
Single service	66601	51	9684	236
Single service	1303031	518	963	1001
Two services	11	43	17025	670
Two services	541	41	12005	665
Two services	66601	42	11769	613
Two services	1303031	267	1679	2731
Three services	11	40	12076	679
Three services	541	40	12060	682
Three services	66601	40	12075	664
Three services	1303031	188	2523	2228

First, we observed that there is no difference between isPrime(11) and isPrime(541) regarding average latency for all three scenarios. We believe this is because the backend service is IO bound and not CPU bound. When we increase the prime number, the CPU utilization increases while maintaining all other resources' utilization almost constant. When comparing isPrime(11) with isPrime(541), with 500 concurrency level, both are not sufficient to stress the CPU to its maximum utilization; therefore, additional work added by isPrime(541) did not add latency.

Second, we observed that in low CPU usage (isPrime(11) and isPrime(541)) cases, single-service performance is better than two-service or three-service configurations, with respect to average latency and throughput. For example, for the isPrime(11) test, we observed an average latency of 28ms for the single-service configuration, whereas for two-service and three-service configurations, we observed an average latency of 43ms and 40ms, respectively. We believe this is because of the trade-off between gains due to more nodes and additional latency due to an additional hop. For example, in the isPrime(11) and isPrime(541) tests, the CPU is not fully utilized; therefore, the average latency and throughput are mainly governed by the speed of the network (given that the prime check application has a very little memory footprint). Hence, adding more servers with a load balancer only adds an additional hop, and the load balancer has to do twice as much IO as the backend server (adding more servers does not improve the response time from a backend service). This shows that when scaling a system, we should first identify the limiting factor and then scale that resource. Blindly scaling a system with many servers will degrade performance.

Third, we observed that when CPU usage is high (isPrime(66601)), there is a performance gain in two-service and three-service setups. However, the percentage performance gain (average latency and throughput) is small (increase the number of services by two and throughput increases by a factor much less than two). For example, in the isPrime(66601) case, we get a percentage throughput increase of 1.21 at two services and 1.24 at three services.

Also, we observed that when CPU usage is very high (isPrime(1303031)), there is a speed up close to x, where x is the number of services. For example, in the isPrime(1303031) test, we noted a percentage throughput increase of 1.74 and 2.62. We believe this is because the cost and gains of the load balancer add up positively with CPU-heavy backend services. For example, in the isPrime(66601) test, the CPU utilization is comparatively high, and in the isPrime(1303031) test, CPU utilization is even higher. Hence, we can make an assumption that a single-service configuration CPU is operating at its peak level. When we increase the number of services, the load on a single service decreases. For example, the CPU load decreases in our tests. When the CPU utilization decreases, the queue lengths decrease. Hence, the response times decreases significantly.

In order to obtain the intended return on investment, we should only add resources that are limiting resources (bottlenecks). For example, in these tests, for the low-CPU intensity cases, (isPrime(11) and isPrime(541)), adding more services does not improve performance because the existing resources in the system are not fully utilized. When we increase the CPU intensity to a higher level (so that CPU utilization is very high), we get benefits by scaling the system.

Finally, we observed that there is no difference in 99 percentile latency values between two-service and three-service configurations; however, we noted a significant variance between one-service and two-service configurations. We believe this is because of the number of network hops. For example, the number of network hops per request is two for single service, four for two services and four for three services. In the workload characterization of web servers, it has been shown that network traffic has a high 99 percentile latency. When the number of network links per request increases from two to four, the 99 percentile latency increases. But, when scaling from two services to three services, the number of network links per request remains constant at four. Therefore, the 99 percentile latency is not affected.

9.5 Summary

In this section, we looked at the performance impact of load balancing. We used a prime checking backend service that can simulate different levels of CPU use. By changing the prime number, we tested the performance of four different CPU intensity levels. We showed that for low CPU-bound workloads, adding a load balancer and more server replications do not give a performance gain, and, sometimes, can lead to decreased performance. Also, we observed that when the backend CPU utilization is high, adding more servers with a load balancer gives a performance gain.

These observations are very useful in capacity planning. We often try to improve the performance of a computer system by adding more resources with a load balancer. As we have shown in this section, adding more resources sometimes degrades performance. Hence, a more general guideline to capacity planning should include checking the backend server's utilization to make sure it is fully utilized. If the backend service is lightly loaded before adding more backend servers, adding more servers will often degrade system performance.

10. CONCLUSION

Due to the wide adoption of Server based systems, understanding the performance of server based systems, under different conditions is important. In this research, we characterized the web server systems under different configurations. We first presented a summary of prevalent server architectures. Second, we provided a systematic approach for performance testing, and presented a novel Python open source library for latency analysis. Third, we experimented on existing server architectures, and proposed eight new server architectures. Our analysis shows that under different conditions the new architectures outperform the existing architectures. Fourth, we did an extensive tail latency analysis of Java microservices. Fifth, we explored the scalability characteristics of web servers. Sixth, we proposed a novel approach to model the closed system performance using discrete event simulation. Finally, we showed that unless used carefully, load balancing decreases the performance of server based systems.

In summary we make the following contributions in this research

1. Implemented a novel open source library for workload characterization
2. Proposed a systematic approach for performance testing of web servers
3. Proposed eight new server architectures
4. Discussed the hardware, software implications for server performance
5. Performed an extensive tail latency analysis of microservices
6. Identified the scalability characteristics of middleware using Universal law of scalability
7. Proposed a novel approach to model web server closed system performance using discrete event simulation
8. Identified the impact of load balancing for server based systems.

We explored several weaknesses in our novel server architectures, and proved the claims using hardware and software performance counters. These lead to further improvements of the novel server architectures. We expect to explore them in the future.

Discrete event simulation for half open systems are still unknown. Also, employing queues and processing elements at different layers in the OSI model are promising future works.

REFERENCES

[1] B. Erb, "Concurrent Programming for Scalable Web Architectures", Diploma, Institute of Distributed Systems Faculty of Engineering and Computer Science Ulm University, 2012.

[2] "XML-RPC Specification", Xmlrpc.scripting.com, 2019. [Online]. Available: http://xmlrpc.scripting.com/spec.html. [Accessed: 24- Jun- 2019].

[3] "SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)", W3.org, 2019. [Online]. Available: https://www.w3.org/TR/soap12-part1/. [Accessed: 24- Jun- 2019].

[4] “WSDL Specification ”, 2019. [Online]. Available: https://www.w3.org/TR/2007/REC-wsdl20-20070626/. [Accessed: 24- Jun- 2019].

[5] "OASIS UDDI Specification TC | OASIS", Oasis-open.org, 2019. [Online]. Available:https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=uddi-spec. [Accessed: 24- Jun- 2019].

[6] F. Royomas, “Architectural styles and the design of network-based software architectures”, Ph.D. thesis, University of California, Irvine (2000)

[7] p. Roy and S. Hafidi, Concepts, techniques and models of computer programming. New Delhi: PHI Learning Private Ltd., 2009.

[8] B. Cantrill and J. Bonwick, "Real-world Concurrency", Queue, vol. 6, no. 5, p. 16, 2008. Available: 10.1145/1454456.1454462.

[9] R. Arnon. Gosling, James and L. Deutsch, “Fallacies of Distributed Computing Explained”, Tech. Rep., Sun Microsystems (2006)

[10] G. Debasish, S. Justin, T. Kresten and V. Steve, “Programming language impact on the development of distributed systems”, Journal of Internet Services and Applications (2011), vol. Issue 2 / 2011:pp. 1–8, 10.1007/s13174-011-0042-y

[11] I. WSO2, "Cloud Native Programming Language", Ballerina.io, 2019. [Online]. Available: https://ballerina.io/. [Accessed: 24- Jun- 2019].

[12] K. Dan, “C10K problem”, Tech. Rep., Kegel.com (2006)

[13] O. John, “Why reads are a Bad Idea (for most purposes)”, in USENIX Winter Technical Conference

[14] V. Behren, R. Condit, Jeremy, B. Eric, “Why events are a bad idea (for high-concurrency servers), in: Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9, USENIX Association, Berkeley, CA, USA, pp. 4–4

[15] M. Welsh, D. Culler, and B. Eric, “SEDA: an architecture for well conditioned, scalable internet services”, in: Proceedings of the eighteenth ACM symposium on Operating systems principles, SOSP ’01, ACM, New York, NY, USA,pp. 230–243

[16] W. Stevens,.F. Richard, R. Bill and M. Andrew, “Unix Network Programming”, Volume 1: e Sockets Networking API (3rd Edition), Addison-Wesley Professional (2003)

[17] S. Vivek, P. Druschel, W. Zwaenepoel ,”Flash: an efficient and portable web server, in: Proceedings of the annual conference on USENIX Annual Technical Conference, USENIX Association, Berkeley, CA, USA, pp. 15–15

[18] M. Welsh, “A Retrospective on SEDA, Blog Post”, http://mattwelsh.blogspot.com/2010/07/retrospective-on-seda.html (2010)

[19] Lmax-exchange.github.io, 2019. [Online]. Available: https://lmax-exchange.github.io/disruptor/files/Disruptor-1.0.pdf. [Accessed: 24-Jun- 2019].

[20] C. Hewitt, P. Bishop, and R. Steiger, ”A universal modular ACTOR formalism for artificial intelligence, in: Proceedings of the 3rd international joint conference on Artificial intelligence, IJCAI’73, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 235–245

[21] C. Hoare, “Communicating sequential processes”. Commun. ACM (1978), vol. 21(8):pp. 666–677

[22] M. Mazzara and B. Meyer, Present and ulterior software engineering.
[s.l.]: Springer, PU, 2017.

[23] T. Salah, J. Zemerly, C. Yeun, M. Al-Qutayri and Y. Al-Hammadi, "The evolution of distributed systems towards microservices architecture", in 11th International Conference for Internet Technology and Secured Transactions, 2016.

[24] V. Pacheco, Microservice patterns and best practices [s.3.], PL, 2013..

[25] A. Balalaie, A. Heydarnoori and P. Jamshidi, "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture", IEEE Software, vol. 33, no. 3, pp. 42-52, 2016. Available: 10.1109/ms.2016.64.

[26] M. Villamizar, O. GarcÃl’s, H. Castro, M. Verano, L. Salamanca and R.Casallas, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud", in 10th Computing Colombian Conference, 2015.

[27] R. Heinrich et al., "Performance Engineering for Microservices: Research Challenges and Directions", in Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion, 2017.

[28] "TPC-W - Homepage", 2019. [Online]. Available: http://www.tpc.org/tpcw/. [Accessed: 13- Mar- 2019].

[29] "SPECjvm2008", Spec.org, 2019. [Online]. Available: https://www.spec.org/jvm2008/. [Accessed: 13- Mar- 2019].

[30] "JPetStore Demo", Jpetstore.cfapps.io, 2019. [Online]. Available: https://jpetstore.cfapps.io/. [Accessed: 13- Mar- 2019].

[31] C. Aderaldo, N. Mendonca, C. Pahl and P. Jamshidi, "Benchmark Requirements for Microservices Architecture Research", in IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering, 2017.

[32] "Acme Air", GitHub, 2019. [Online]. Available: https://github.com/acmeair/. [Accessed: 13- Mar- 2019].

[33]"Spring cloud demo apps" 2019. [Online]. Available: ttps://github.com/kbastani/spring-cloudmicroservices-example. [Accessed: 13- Mar- 2019].

[34] "Microservices Demo: Sock Shop", Microservices-demo.github.io, 2019. [Online]. Available: https://microservices-demo.github.io. [Accessed:13- Mar- 2019].

[35]"aspnet/MusicStore", GitHub, 2019. [Online]. Available:https://github.com/aspnet/MusicStore. [Accessed: 13- Mar- 2019].

[36] A. Sriraman and T. Wenisch, " M Suite: A Benchmark Suite for Microservices", in IEEE International Symposium on Workload Characterization, 2018.

[37] A. Camargo, I. Salvadori, R. Mello and F. Siqueira, "An architecture to automate performance tests on microservices", in Proceedings of the 18th International Conference on Information Integration and Web-based Applications and Services, 2016.

[38] T. Ueda, T. Nakaike and M. Ohara, "Workload characterization for microservices", in IEEE International Symposium on Workload Characterization, 2016.

[39] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar and M. Steinder, "Performance Evaluation of Microservices Architectures Using Containers", in NCA ’15 Proceedings of the 2015 IEEE 14th International Symposium on Network Computing and Applications (NCA),
2015.

[40] L. Ismail, D. Hagimont, and J. Mossi'ere, "Evaluation of the mobile agents technology: Comparison with the client/server paradigm," Information Science and Technology (1ST) , vol. 19, 2000.

[41] W. A. De Vries and R. A. Fleck, "Client/server infrastructure: a case study in planning and conversion," Industrial Management & Data Systems, vol. 97, no, 6, pp, 222-232, 1997

[42] K. Kulesza, Z. Kotulski, and K. Kulesza, "On mobile agents resistance to traffic analysis," Electronic Notes in Theoretical Computer Science, vol. 142, pp. 181-193, 2006

[43] S. Newman, Building Microservices. " O'Reilly Media, Inc.", 2015

[44] Wen, Y. Ma, and X. Chen, "ESB infrastructure's autonomous mechanism of SOA," in 2009 International Symposium on Intelligent Ubiquitous Computing and Education. IEEE, 2009, pp. 13 -17.

[45] Y. Sun, S. Nanda, and T. Jaeger, "Security-as-a-service for Microservices based cloud applications," in 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (Cloud Corn). IEEE, 2015, pp. 50-5 7.

[46] 2015, pp. 50-5 7. [28] Hassan, M., Zhao, W., & Yang, J. (2010, July). Provisioning web services from resource constrained mobile devices. In Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on (pp. 490-497). IEEE.

[47] Namiot, D., & Sneps-Sneppe, M. (2014). On Micro-services Architecture. International Journal of Open Information Technologies, 2(9), 24-27.

[48] "Apache JMeter - Apache JMeter", JMeter.apache.org, 2019. [Online].
Available: https://JMeter.apache.org/. [Accessed: 13- Mar- 2019].

[49] "Creating a pagerank analytics platform using Spring Boot microservices," http://www.kennybastani.com/2016/01/spring-boot-graphprocessing-microservices.html, 2016, [Online; accessed 18-January2017]

[50] "MusicStore – steeltoeoss samples," https://github.com/SteeltoeOSS/Samples/tree/master/MusicStore, 2017, [Online; accessed 18-January-2017]

[51]"SPECweb 2009 Benchmark", Spec.org, 2019. [Online]. Available: https://www.spec.org/web2009/. [Accessed: 24- Jun- 2019].

[52]"TPC-C - Homepage", Tpc.org, 2019. [Online]. Available: http://www.tpc.org/tpcc/. [Accessed: 24- Jun- 2019].

[53]"SPECjEnterprise®2010", Spec.org, 2019. [Online]. Available: https://www.spec.org/jEnterprise2010/. [Accessed: 24- Jun- 2019].

[54] B. Schroeder, A. Wierman and M. Harchol-Balter, "Closed versus open system models and their impact on performance and scheduling", in Symposium on Networked Systems Design and Implementation (NSDI), 2006

[55] "PasinduTennage/python-latency-analysis", GitHub, 2019. [Online]. Available: https://github.com/PasinduTennage/python-latency-analysis. [Accessed: 24- Jun- 2019].

[56] Sun Microsystems. RPC: Remote Procedure Call Protocol Specification Version 2. Internet Network Working Group RFC1057, June 1988.

[57] Sun Microsystems, Inc. Java Remote Method Invocation. http://java.sun.com/products/jdk/rmi/ .

[58] Apache Software Foundation. The Apache web server. http://www. Apache.org

[59]Microsoft Corporation. IIS 5.0 Overview. http://www.microsoft.com/windows2000/library/howitworks/
iis/iis5techove%rview.asp

[60] M. Thompson, D. Gregory, M. Farley, P. Barker and A. Stewart. "Disruptor : High performance alternative to bounded queues for exchanging data between concurrent threads." (2011).

[61] "PasinduTennage/server-architectures", GitHub, 2019. [Online]. Available: https://github.com/PasinduTennage/server-architectures/. [Accessed: 13- Mar- 2019]

[62]Imysql.com, 2019. [Online]. Available: http://imysql.com/wpcontent/uploads/2014/10/sysbench-manual.pdf. [Accessed: 13-Mar2019].

[63] S. Lehrig, R. Sanders, G. Brataas, M. Cecowski, S. Ivansek and J. Polutnik, "CloudStore - towards scalability, elasticity, and efficiency benchmarking and analysis in Cloud computing", Future Generation Computer Systems, vol. 78, pp. 115-126, 2018. Available: 10.1016/j.future.2017.04.018.

[64] "PasinduTennage/GC-Perfomance", GitHub, 2019. [Online]. Available: https://github.com/PasinduTennage/GC-Perfomance/. [Accessed: 13- Mar- 2019].

[65] "Integration - On-Premise and in the Cloud", Wso2.com, 2019. [Online]. Available: https://wso2.com/integration/. [Accessed: 24- Jun- 2019].

[66] "wso2/performance-common", GitHub, 2019. [Online]. Available: https://github.com/wso2/performance-common. [Accessed: 13- Mar2019].

[67] "Ubuntu Manpage: sar - Collect, report, or save system activity information.", Manpages.ubuntu.com, 2019. [Online]. Available: http://manpages.ubuntu.com/manpages/cosmic/man1/sar.sysstat.1.html. [Accessed: 13- Mar- 2019].

[68] Man7.org. (2019). perf(1) - Linux manual page. [online] Available at: http://man7.org/linux/man-pages/man1/perf.1.html [Accessed 24 Jun. 2019].

[69] "Ubuntu Manpage: iftop - display bandwidth usage on an interface by host", Manpages.ubuntu.com, 2019. [Online]. Available: http://manpages.ubuntu.com/manpages/bionic/man8/iftop.8.html. [Accessed: 13- Mar- 2019].

[70] "Server Architecture Test Results.xlsx", Google Docs, 2019. [Online]. Available: https://drive.google.com/file/d/1PDULdT83xCCxHsMGmn5Pl_gZqW-IqYwx/view?usp=sharing. [Accessed: 24- Jun- 2019].

[71]T. Brecht, E. Arjomandi, C. Li and H. Pham, "Controlling garbage collection and heap growth to reduce the execution time of Java applications", ACM SIGPLAN Notices, vol. 36, no. 11, pp. 353-366, 2001. Available: 10.1145/504311.504308.

[72] S. Blackburn, P. Cheng and K. McKinley, "Myths and realities", ACM SIGMETRICS Performance Evaluation Review, vol. 32, no. 1, p. 25, 2004. Available: 10.1145/1012888.1005693.

[73] L. Gidra, G. Thomas, J. Sopena and M. Shapiro, "A study of the scalability of stop-the-world garbage collectors on multicores", ACM SIGPLAN Notices, vol. 48, no. 4, p. 229, 2013. Available: 10.1145/2499368.2451142.

[74] M. Carpen-Amarie, P. Marlier, P. Felber and G. Thomas, "A performance study of Java garbage collectors on multicore architectures", in . In: Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores, 2015.

[75] J. Thönes, “microservices," IEEE Software, Vol. 32, Issue. 1, pp. 113-116.

[76] "Spring Projects", Spring.io, 2019. [Online]. Available: https://spring.io/projects/spring-boot. [Accessed: 13- Mar- 2019]

[77] "PasinduTennage/springboot-test", GitHub, 2019. [Online]. Available: https://github.com/PasinduTennage/springboot-test. [Accessed: 13- Mar2019].

[78]"microservices-demo/load-test", GitHub, 2019. [Online]. Available: https://github.com/microservices-demo/loadtest/blob/master/locustfile.py. [Accessed: 13- Mar- 2019].

[79] "PasinduTennage/socksshopJMeter", GitHub, 2019. [Online]. Available: https://github.com/PasinduTennage/socksshopJMeter. [Accessed: 13- Mar- 2019].

[80] "Amazon EC2 Instance Types - Amazon Web Services", Amazon Web Services, Inc., 2019. [Online]. Available: https://aws.amazon.com/ec2/instance-types/. [Accessed: 13- Mar2019].

[81] "Overview - SimPy 3.0.11 documentation", Simpy.readthedocs.io, 2019. [Online]. Available: https://simpy.readthedocs.io/en/latest/. [Accessed: 13- Mar- 2019].

[82] J. Li, N. Sharma, D. Ports and S. Gribble, "Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency", in Proceedings of the ACM Symposium on Cloud Computing, 2014.

[83] "PasinduTennage/microservices-descrete-event-simulation", GitHub, 2019. [Online]. Available:https://github.com/PasinduTennage/microservices-descrete-eventsimulation. [Accessed: 13- Mar- 2019].

[84] "PasinduTennage/micro-services-tail-index-analysis-results", GitHub, 2019. [Online]. Available: https://github.com/PasinduTennage/microservices-tail-index-analysis-results. [Accessed: 13- Mar- 2019].

[85] J. Hennessy and D. Patterson, Computer architecture, 6th edition, 2017

[86] N. Gunther, Guerrilla capacity planning. Berlin: Springer, 2011.

[87] Amazon.com, 2019. [Online]. Available: https://www.amazon.com/Performance-Modeling-Design-Computer-Systems/dp/1107027500. [Accessed: 24- Jun- 2019].

[88] "What is NGINX? - NGINX", NGINX, 2019. [Online]. Available: https://www.nginx.com/resources/glossary/nginx/. [Accessed: 24- Jun- 2019].

[89] P. Tennage, S. Perera, M. Jayasinghe and S. Jayasena, "An Analysis of Holistic Tail Latency Behaviors of Java Microservices," 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), Zhangjiajie, China, 2019, pp. 697-705, doi: 10.1109/HPCC/SmartCity/DSS.2019.00104.

Appendix A: Server Architecture Results

Use Case	Backend Architecture	Heap	Concurrency	Workload	Average Latency	Throughput	Percentile 99
io	Blocking	2g	300	1024	47.25204112	6334.343333	450
io	Blocking	2g	10	1024	1.614496805	5895.805	3
io	Blocking	100m	300	1024	47.19426614	6341.978333	450
io	Blocking	100m	10	1024	1.621038504	5876.463333	3
io	Blocking	2g	300	10	43.25755656	6909.336667	1025
io	Blocking	2g	10	10	1.258362928	7610.721667	4
io	Blocking	100m	300	10	42.65937912	7008.3	1026
io	Blocking	100m	10	10	1.265719998	7565.363333	5
io	Blocking Disruptor	2g	300	1024	60.12030311	4985.13	1041
io	Blocking Disruptor	2g	10	1024	2.038677007	4700.855	4
io	Blocking Disruptor	100m	300	1024	55.82458767	5361.035	1038
io	Blocking Disruptor	100m	10	1024	2.05127292	4675.47	4
io	Blocking Disruptor	2g	300	10	45.25049488	6615.258333	1029
io	Blocking Disruptor	2g	10	10	1.659779774	5796.76	4
io	Blocking Disruptor	100m	300	10	48.83835211	5907.48	1030
io	Blocking Disruptor	100m	10	10	1.667509315	5766.868333	4
io	Blocking Actor	2g	300	1024	66.589535	4495.97	1029
io	Blocking Actor	2g	10	1024	1.661425391	5739.803333	3
io	Blocking Actor	100m	300	1024	70.4562375	4249.566667	1046
io	Blocking Actor	100m	10	1024	1.660005594	5743.82	3
io	Blocking Actor	2g	300	10	125.697167	2382.946667	1181
io	Blocking Actor	2g	10	10	1.27070197	7546.645	9
io	Blocking Actor	100m	300	10	116.6674021	2567.8	1174

io	Blocking Actor	100m	10	10	1.282053627	7473.543333	9
io	NIO	2g	300	1024	77.38879295	3868.991667	1239
io	NIO	2g	10	1024	2.493246301	3859.613333	3
io	NIO	100m	300	1024	75.08709919	3988.651667	1237
io	NIO	100m	10	1024	2.469132083	3899.95	3
io	NIO	2g	300	10	65.07476999	4342.718333	1235
io	NIO	2g	10	10	2.422317976	4011.46	9
io	NIO	100m	300	10	61.84589727	4400.646667	1067
io	NIO	100m	10	10	2.753854226	3538.756667	10
io	NIO Disruptor	2g	300	1024	222.2785216	1348.303333	225
io	NIO Disruptor	2g	10	1024	7.455909614	1318.05	10
io	NIO Disruptor	100m	300	1024	222.6888782	1346.171667	226
io	NIO Disruptor	100m	10	1024	7.419150765	1324.368333	10
io	NIO Disruptor	2g	300	10	222.8983815	1345.195	226
io	NIO Disruptor	2g	10	10	7.370149589	1336.441667	10
io	NIO Disruptor	100m	300	10	219.1836567	1367.96	222
io	NIO Disruptor	100m	10	10	7.380497187	1334.508333	10
io	nio.netty	2g	300	1024	45.61846781	6560.621667	1026
io	nio.netty	2g	10	1024	1.583609054	6017.57	3
io	nio.netty	100m	300	1024	44.95371866	6656.175	1023
io	nio.netty	100m	10	1024	1.631862682	5840.95	3
io	nio.netty	2g	300	10	46.31986796	6425.781667	1014
io	nio.netty	2g	10	10	2.975211939	3291.436667	13
io	nio.netty	100m	300	10	44.29790401	6737.075	893
io	nio.netty	100m	10	10	2.76077569	3543.083333	12
io	NIO Actor	2g	300	1024	219.9928153	1362.395	222
io	NIO Actor	2g	10	1024	7.402816905	1326.988333	8
io	NIO Actor	100m	300	1024	220.0707037	1362.116667	222
io	NIO Actor	100m	10	1024	7.53036429	1305.096667	9
io	NIO Actor	2g	300	10	219.4570582	1366.325	221
io	NIO Actor	2g	10	10	7.530835706	1309.331667	9
io	NIO Actor	100m	300	10	223.7686258	1340.008333	225
io	NIO Actor	100m	10	10	7.472770738	1318.373333	9

io	SEDA Actor	2g	300	1024	52.41849097	5713.15	1038
io	SEDA Actor	2g	10	1024	1.890702227	5075.355	16
io	SEDA Actor	100m	300	1024	50.74854106	5898.981667	1037
io	SEDA Actor	100m	10	1024	1.860883214	5156.435	14
io	SEDA Actor	2g	300	10	53.0544053	5620.408333	1036
io	SEDA Actor	2g	10	10	2.052792265	4726.45	15
io	SEDA Actor	100m	300	10	49.88658862	5995.356667	1039
io	SEDA Actor	100m	10	10	1.908247946	5072.656667	13
io	SEDA Disruptor	2g	300	1024	214.231783	1397.981667	1301
io	SEDA Disruptor	2g	10	1024	26.50623424	356.2316667	168
io	SEDA Disruptor	100m	300	1024	315.2990178	950.1216667	1681
io	SEDA Disruptor	100m	10	1024	30.51067582	320.35	181
io	SEDA Disruptor	2g	300	10	270.9059618	1102.92	1534
io	SEDA Disruptor	2g	10	10	27.06932625	349.3866667	180
io	SEDA Disruptor	100m	300	10	354.9513499	843.78	1849
io	SEDA Disruptor	100m	10	10	29.5695385	319.715	183
io	SEDA Queue	2g	300	1024	51.74499456	5711.538333	1037
io	SEDA Queue	2g	10	1024	1.312791451	4084.446667	2
io	SEDA Queue	100m	300	1024	56.26819309	5239.421667	1046
io	SEDA Queue	100m	10	1024	1.348066278	4006.478333	3
io	SEDA Queue	2g	300	10	47.97686453	5957.086667	1038
io	SEDA Queue	2g	10	10	1.318392991	5833.943333	3
io	SEDA Queue	100m	300	10	50.50901413	5839.166667	1044
io	SEDA Queue	100m	10	10	1.157056782	5022.461667	3
io	NIO2	2g	300	1024	104.9787417	2823.525	476
io	NIO2	2g	10	1024	2.231680448	4320.088333	3
io	NIO2	100m	300	1024	98.44197418	3008.905	496
io	NIO2	100m	10	1024	2.235508971	4311.201667	3
io	NIO2	2g	300	10	109.0944621	2710.081667	504
io	NIO2	2g	10	10	2.385472207	4064.21	3
io	NIO2	100m	300	10	104.4051619	2823.916667	552
io	NIO2	100m	10	10	2.393698387	4046.371667	3
io	NIO2 Actor	2g	300	1024	218.206772	1373.59	222

io	NIO2 Actor	2g	10	1024	7.300048074	1345.14	8
io	NIO2 Actor	100m	300	1024	217.3206885	1379.485	222
io	NIO2 Actor	100m	10	1024	7.295037464	1346.388333	8
io	NIO2 Actor	2g	300	10	216.8088632	1383.015	218
io	NIO2 Actor	2g	10	10	7.49558239	1316.7	8
io	NIO2 Actor	100m	300	10	221.1185063	1356.088333	222
io	NIO2 Actor	100m	10	10	7.467339722	1320.181667	8
io	NIO2 Disruptor	2g	300	1024	217.1027458	1380.753333	226
io	NIO2 Disruptor	2g	10	1024	7.407343018	1326.793333	10
io	NIO2 Disruptor	100m	300	1024	216.6023524	1383.951667	227
io	NIO2 Disruptor	100m	10	1024	7.426707164	1323.003333	10
io	NIO2 Disruptor	2g	300	10	219.3027871	1367.198333	230
io	NIO2 Disruptor	2g	10	10	7.416778905	1327.778333	10
io	NIO2 Disruptor	100m	300	10	215.8492122	1389.093333	224
io	NIO2 Disruptor	100m	10	10	7.427054933	1326.523333	10
cpu	Blocking	2g	300	27059	41.95057963	7127.661667	1026
cpu	Blocking	2g	10	27059	1.41473609	6779.206667	6
cpu	Blocking	100m	300	27059	42.53772236	7031.351667	1027
cpu	Blocking	100m	10	27059	1.429942939	6704.808333	6
cpu	Blocking	2g	300	11	41.61693832	7185.248333	1025
cpu	Blocking	2g	10	11	1.253746778	7642.678333	3
cpu	Blocking	100m	300	11	42.1766678	7082.888333	1025
cpu	Blocking	100m	10	11	1.265173866	7571.598333	5
cpu	Blocking Disruptor	2g	300	27059	49.59746153	6034.996667	1032
cpu	Blocking Disruptor	2g	10	27059	2.001308216	4822.09	4
cpu	Blocking Disruptor	100m	300	27059	48.18726793	6211.376667	1033
cpu	Blocking Disruptor	100m	10	27059	2.019313812	4783.623333	4
cpu	Blocking Disruptor	2g	300	11	45.95102757	6511.248333	1029
cpu	Blocking Disruptor	2g	10	11	1.670978532	5758.915	4

cpu	Blocking Disruptor	100m	300	11	46.28904181	6463.263333	1029
cpu	Blocking Disruptor	100m	10	11	1.677011875	5732.981667	4
cpu	Blocking Actor	2g	300	27059	61.23965546	4890.34	1096
cpu	Blocking Actor	2g	10	27059	1.310621717	7315.468333	10
cpu	Blocking Actor	100m	300	27059	55.17637527	5429.465	1046
cpu	Blocking Actor	100m	10	27059	1.327115442	7228.335	10
cpu	Blocking Actor	2g	300	11	111.827449	2678.58	1171
cpu	Blocking Actor	2g	10	11	1.266810457	7561.166667	9
cpu	Blocking Actor	100m	300	11	107.077582	2797.148333	1167
cpu	Blocking Actor	100m	10	11	1.285105749	7455.818333	9
cpu	NIO	2g	300	27059	80.50125035	3698.963333	1449
cpu	NIO	2g	10	27059	2.849994063	3410.731667	4
cpu	NIO	100m	300	27059	76.44050117	3910.076667	1445
cpu	NIO	100m	10	27059	2.438170855	3980.148333	4
cpu	NIO	2g	300	11	62.63000496	4367.288333	1228
cpu	NIO	2g	10	11	2.450226367	3967.378333	9
cpu	NIO	100m	300	11	62.10817903	4473.633333	1047
cpu	NIO	100m	10	11	2.623294613	3712.99	9
cpu	NIO Disruptor	2g	300	27059	222.7917755	1345.783333	225
cpu	NIO Disruptor	2g	10	27059	7.360839116	1337.678333	9
cpu	NIO Disruptor	100m	300	27059	219.6892039	1364.78	225
cpu	NIO Disruptor	100m	10	27059	7.464027541	1319.25	10
cpu	NIO Disruptor	2g	300	11	222.9030043	1345.111667	226
cpu	NIO Disruptor	2g	10	11	7.412936035	1328.305	10
cpu	NIO Disruptor	100m	300	11	219.0704957	1368.618333	222
cpu	NIO Disruptor	100m	10	11	7.379793255	1334.818333	10
cpu	nio.netty	2g	300	27059	44.23255581	6743.721667	1011
cpu	nio.netty	2g	10	27059	2.723406408	3590.683333	13
cpu	nio.netty	100m	300	27059	43.79176838	6814.983333	1012
cpu	nio.netty	100m	10	27059	2.656892085	3681.203333	13
cpu	nio.netty	2g	300	11	46.30761705	6434.64	1010
cpu	nio.netty	2g	10	11	3.008830878	3256.376667	13

cpu	nio.netty	100m	300	11	44.82422817	6654.346667	901
cpu	nio.netty	100m	10	11	2.98788177	3278.531667	13
cpu	NIO Actor	2g	300	27059	220.176354	1361.796667	225
cpu	NIO Actor	2g	10	27059	7.539217225	1306.505	9
cpu	NIO Actor	100m	300	27059	219.7191533	1364.653333	221
cpu	NIO Actor	100m	10	27059	7.531401009	1307.866667	9
cpu	NIO Actor	2g	300	11	223.3663803	1342.448333	225
cpu	NIO Actor	2g	10	11	7.571894124	1302.088333	9
cpu	NIO Actor	100m	300	11	219.1776658	1368.065	221
cpu	NIO Actor	100m	10	11	7.457694817	1320.661667	9
cpu	SEDA Actor	2g	300	27059	47.70376547	6247.071667	1029
cpu	SEDA Actor	2g	10	27059	1.794509583	5390.665	11
cpu	SEDA Actor	100m	300	27059	46.84409358	6382.675	1034
cpu	SEDA Actor	100m	10	27059	2.008573827	4825.15	12
cpu	SEDA Actor	2g	300	11	55.85708172	5335.52	1036
cpu	SEDA Actor	2g	10	11	1.990665332	4875.91	13
cpu	SEDA Actor	100m	300	11	50.78160474	5891.573333	1039
cpu	SEDA Actor	100m	10	11	1.824942505	5296.821667	13
cpu	SEDA Disruptor	2g	300	27059	283.0924037	1056.451667	1581
cpu	SEDA Disruptor	2g	10	27059	25.63170368	367.0866667	144
cpu	SEDA Disruptor	100m	300	27059	384.2951838	780.2766667	2134.35
cpu	SEDA Disruptor	100m	10	27059	29.07306338	339.2716667	162
cpu	SEDA Disruptor	2g	300	11	269.3688588	1110.736667	1486
cpu	SEDA Disruptor	2g	10	11	26.93552676	358.495	170
cpu	SEDA Disruptor	100m	300	11	350.4485533	854.4766667	1853
cpu	SEDA Disruptor	100m	10	11	30.46472607	316.57	188
cpu	SEDA Queue	2g	300	27059	52.44140238	5670.68	1041
cpu	SEDA Queue	2g	10	27059	2.094543423	4567.76	5
cpu	SEDA Queue	100m	300	27059	58.64852046	5069.256667	1049
cpu	SEDA Queue	100m	10	27059	2.22345567	4190.301667	6
cpu	SEDA Queue	2g	300	11	47.1127137	6162.146667	1037
cpu	SEDA Queue	2g	10	11	1.28204123	5820.313333	3
cpu	SEDA Queue	100m	300	11	50.56040038	5827.405	1044

cpu	SEDA Queue	100m	10	11	1.140802349	5065.043333	2
cpu	NIO2	2g	300	27059	117.2916844	2514.138333	494
cpu	NIO2	2g	10	27059	2.246075444	4306.538333	4
cpu	NIO2	100m	300	27059	103.8334423	2847.861667	517
cpu	NIO2	100m	10	27059	2.265022354	4277.708333	4
cpu	NIO2	2g	300	11	107.9167758	2739.768333	532
cpu	NIO2	2g	10	11	2.379587755	4069.665	3
cpu	NIO2	100m	300	11	105.0399429	2815.896667	553
cpu	NIO2	100m	10	11	2.395911577	4046.303333	3
cpu	NIO2 Actor	2g	300	27059	217.3721299	1379.33	221
cpu	NIO2 Actor	2g	10	27059	7.299981337	1348.511667	8
cpu	NIO2 Actor	100m	300	27059	217.7210193	1377.078333	221
cpu	NIO2 Actor	100m	10	27059	7.300840766	1348.77	8
cpu	NIO2 Actor	2g	300	11	220.6770699	1358.761667	222
cpu	NIO2 Actor	2g	10	11	7.495656335	1316.668333	8
cpu	NIO2 Actor	100m	300	11	216.7076451	1383.723333	218
cpu	NIO2 Actor	100m	10	11	7.468013434	1320.033333	8
cpu	NIO2 Disruptor	2g	300	27059	215.8142028	1389.166667	223
cpu	NIO2 Disruptor	2g	10	27059	7.401870705	1329.98	10
cpu	NIO2 Disruptor	100m	300	27059	217.7303616	1377.041667	228
cpu	NIO2 Disruptor	100m	10	27059	7.378452228	1334.585	10
cpu	NIO2 Disruptor	2g	300	11	215.7898692	1389.46	224
cpu	NIO2 Disruptor	2g	10	11	7.399554304	1331.25	10
cpu	NIO2 Disruptor	100m	300	11	218.0454974	1375.06	232
cpu	NIO2 Disruptor	100m	10	11	7.315622473	1345.721667	10
memory	Blocking Disruptor	2g	300	1000	57.38226122	4961.973333	1032
memory	Blocking Disruptor	2g	10	1000	2.058496138	4691.876667	4
memory	Blocking Disruptor	100m	300	1000	48.20441609	6215.9	1032
memory	Blocking Disruptor	100m	10	1000	2.126659063	4538.956667	5
memory	Blocking Disruptor	2g	300	10	45.47019469	6577.183333	1029

memory	Blocking Disruptor	2g	10	10	1.668440935	5772.646667	4
memory	Blocking Disruptor	100m	300	10	44.72179648	6689.886667	1030
memory	Blocking Disruptor	100m	10	10	1.676153966	5741.36	4
memory	NIO Disruptor	2g	300	1000	222.7781791	1345.71	226
memory	NIO Disruptor	2g	10	1000	7.491363329	1314.356667	10
memory	NIO Disruptor	100m	300	1000	222.9036318	1344.876667	226
memory	NIO Disruptor	100m	10	1000	7.486477652	1315.476667	10
memory	NIO Disruptor	2g	300	10	222.9440996	1344.653333	225
memory	NIO Disruptor	2g	10	10	7.364011886	1337.126667	10
memory	NIO Disruptor	100m	300	10	222.1136343	1349.71	226
memory	NIO Disruptor	100m	10	10	7.50402936	1312.45	10
memory	SEDA Disruptor	2g	300	1000	279.6512118	1071.433333	1489
memory	SEDA Disruptor	2g	10	1000	25.50148368	381.9333333	144
memory	SEDA Disruptor	100m	300	1000	429.931371	694.6533333	2363
memory	SEDA Disruptor	100m	10	1000	28.31486146	334.8033333	157
memory	SEDA Disruptor	2g	300	10	263.1137873	1135.656667	1474
memory	SEDA Disruptor	2g	10	10	26.5646119	342.9966667	161
memory	SEDA Disruptor	100m	300	10	357.6517459	837.3866667	1974.85
memory	SEDA Disruptor	100m	10	10	29.89491636	316.6366667	177
memory	NIO2 Disruptor	2g	300	1000	217.9252634	1375.676667	227
memory	NIO2 Disruptor	2g	10	1000	7.300700286	1348.02	10
memory	NIO2 Disruptor	100m	300	1000	219.7771736	1364.036667	230
memory	NIO2 Disruptor	100m	10	1000	7.447452073	1322.316667	10
memory	NIO2 Disruptor	2g	300	10	218.6954218	1370.846667	228
memory	NIO2 Disruptor	2g	10	10	7.302063804	1347.673333	10
memory	NIO2 Disruptor	100m	300	10	215.7715067	1389.45	228
memory	NIO2 Disruptor	100m	10	10	7.323086156	1344.11	10
memory	Blocking	2g	300	1000	42.64624893	7011.116667	1026
memory	Blocking	2g	10	1000	1.478129265	6493.99	5
memory	Blocking	100m	300	1000	42.87363651	6981.183333	1028
memory	Blocking	100m	10	1000	1.52629881	6302.756667	6

memory	Blocking	2g	300	10	41.67864073	7172.076667	1025
memory	Blocking	2g	10	10	1.254431934	7643.013333	4
memory	Blocking	100m	300	10	43.35887077	6896.716667	1025
memory	Blocking	100m	10	10	1.254498375	7629.976667	5
memory	Blocking Actor	2g	300	1000	57.18545169	5229.016667	1062
memory	Blocking Actor	2g	10	1000	1.33131509	7207.036667	10
memory	Blocking Actor	100m	300	1000	49.01724848	6113.003333	1041
memory	Blocking Actor	100m	10	1000	1.377947892	6960.483333	10
memory	Blocking Actor	2g	300	10	111.4497778	2687.023333	1172
memory	Blocking Actor	2g	10	10	1.26675205	7565.84	9
memory	Blocking Actor	100m	300	10	111.498195	2687.013333	1178
memory	Blocking Actor	100m	10	10	1.276549779	7500.156667	9
memory	NIO	2g	300	1000	80.65254989	3704.34	1450
memory	NIO	2g	10	1000	2.765818798	3513.646667	4
memory	NIO	100m	300	1000	82.04025574	3639.646667	1452
memory	NIO	100m	10	1000	2.59596046	3739.04	4
memory	NIO	2g	300	10	63.65103585	4380.306667	1236
memory	NIO	2g	10	10	2.739827378	3551.116667	10
memory	NIO	100m	300	10	63.0827016	4519.3	1124.11
memory	NIO	100m	10	10	2.699877314	3605.403333	10
memory	nio.netty	2g	300	1000	44.28918859	6736.803333	1008
memory	nio.netty	2g	10	1000	2.92139817	3351.236667	13
memory	nio.netty	100m	300	1000	46.61649371	6393.306667	1019
memory	nio.netty	100m	10	1000	3.012913656	3251.596667	13
memory	nio.netty	2g	300	10	45.65709803	6520.026667	1010
memory	nio.netty	2g	10	10	2.993488616	3270.17	12
memory	nio.netty	100m	300	10	44.96001928	6633.446667	852
memory	nio.netty	100m	10	10	3.003493724	3262.04	13
memory	NIO Actor	2g	300	1000	223.8766281	1339.23	225
memory	NIO Actor	2g	10	1000	7.535887156	1307.06	9
memory	NIO Actor	100m	300	1000	219.6253645	1364.98	222
memory	NIO Actor	100m	10	1000	7.387707116	1332.586667	9
memory	NIO Actor	2g	300	10	219.1936367	1368.043333	221

memory	NIO Actor	2g	10	10	7.431850569	1326.276667	9
memory	NIO Actor	100m	300	10	220.5712611	1359.366667	225
memory	NIO Actor	100m	10	10	7.561912532	1303.263333	9
memory	SEDA Actor	2g	300	1000	48.25100919	6179.893333	1032
memory	SEDA Actor	2g	10	1000	1.779618835	5428.443333	11
memory	SEDA Actor	100m	300	1000	47.82822075	6251.57	1035
memory	SEDA Actor	100m	10	1000	2.065136683	4695.766667	12
memory	SEDA Actor	2g	300	10	57.76178995	5134.74	1034
memory	SEDA Actor	2g	10	10	2.137923132	4537.793333	15
memory	SEDA Actor	100m	300	10	49.61449731	6020.52	1038
memory	SEDA Actor	100m	10	10	1.85018471	5220.806667	13
memory	SEDA Queue	2g	300	1000	53.17559428	5594.126667	1042
memory	SEDA Queue	2g	10	1000	2.144646035	4407.356667	5
memory	SEDA Queue	100m	300	1000	62.05078233	4793.793333	1053
memory	SEDA Queue	100m	10	1000	2.340584305	4032.256667	7
memory	SEDA Queue	2g	300	10	45.26939448	6304.026667	1036
memory	SEDA Queue	2g	10	10	1.253120441	5874.436667	2
memory	SEDA Queue	100m	300	10	50.16397017	5854.52	1044
memory	SEDA Queue	100m	10	10	1.178564427	4999.166667	3
memory	NIO2	2g	300	1000	118.4348793	2491.68	478
memory	NIO2	2g	10	1000	2.245223203	4309.896667	4
memory	NIO2	100m	300	1000	104.1195928	2836.206667	507
memory	NIO2	100m	10	1000	2.283280989	4235.876667	4
memory	NIO2	2g	300	10	106.0849223	2782.623333	509
memory	NIO2	2g	10	10	2.381603309	4066.963333	3
memory	NIO2	100m	300	10	102.9536321	2866.636667	548
memory	NIO2	100m	10	10	2.391047775	4050.613333	3
memory	NIO2 Actor	2g	300	1000	217.9572432	1375.453333	221
memory	NIO2 Actor	2g	10	1000	7.298456634	1348.783333	8
memory	NIO2 Actor	100m	300	1000	216.2855391	1386.08	220
memory	NIO2 Actor	100m	10	1000	7.413493069	1328.583333	8
memory	NIO2 Actor	2g	300	10	216.1211267	1387.39	218
memory	NIO2 Actor	2g	10	10	7.497318848	1315.976667	8

memory	NIO2 Actor	100m	300	10	216.7826268	1383.09	218
memory	NIO2 Actor	100m	10	10	7.322956051	1345.353333	8
db	Blocking Disruptor	2g	300	NA	233.689187	1282.69	1855
db	Blocking Disruptor	2g	10	NA	8.633328468	1141.866667	36
db	Blocking Disruptor	100m	300	NA	238.7980426	1254.39	3031
db	Blocking Disruptor	100m	10	NA	8.78149961	1123.003333	35
db	NIO Disruptor	2g	300	NA	418.8492762	715.68	1117
db	NIO Disruptor	2g	10	NA	15.47579397	641.7133333	47
db	NIO Disruptor	100m	300	NA	420.7102861	712.8066667	1082
db	NIO Disruptor	100m	10	NA	15.62353336	635.5233333	43
db	SEDA Disruptor	2g	300	NA	571.6627365	523.8633333	2379
db	SEDA Disruptor	2g	10	NA	29.65767285	320.22	145
db	SEDA Disruptor	100m	300	NA	583.4391146	512.9033333	3173
db	SEDA Disruptor	100m	10	NA	36.234157	274.5166667	162
db	NIO2 Disruptor	2g	300	NA	457.9816822	655.4633333	2220.62
db	NIO2 Disruptor	2g	10	NA	14.63030723	678.32	37
db	NIO2 Disruptor	100m	300	NA	421.1748972	433.8166667	1689.56
db	Blocking	2g	300	NA	231.3806648	1295.286667	1310
db	Blocking	2g	10	NA	8.034807203	1226.853333	32
db	Blocking	100m	300	NA	234.5933349	1277.3	1482
db	Blocking	100m	10	NA	8.206684223	1201.236667	31
db	Blocking Actor	2g	300	NA	244.1716129	1227.433333	949
db	Blocking Actor	2g	10	NA	8.127702277	1212.403333	33
db	Blocking Actor	100m	300	NA	246.4228971	1216.253333	925
db	Blocking Actor	100m	10	NA	8.23261636	1197.006667	32
db	NIO	2g	300	NA	585.6595774	335.8766667	8095.04
db	NIO	2g	10	NA	29.63652604	335.87	35
db	NIO	100m	300	NA	593.4068349	333.3866667	8879.85
db	NIO	100m	10	NA	29.83828145	333.46	42
db	nio.netty	2g	300	NA	246.1001774	1217.34	1823

db	nio.netty	2g	10	NA	8.165427207	1207.963333	32
db	nio.netty	100m	300	NA	241.9146173	1238.463333	1483
db	nio.netty	100m	10	NA	8.339160002	1183.256667	32
db	NIO Actor	2g	300	NA	418.7870012	715.81	1054
db	NIO Actor	2g	10	NA	15.14102623	655.67	38
db	NIO Actor	100m	300	NA	424.7576495	706.36	895
db	NIO Actor	100m	10	NA	15.19570864	653.25	35
db	SEDA Actor	2g	300	NA	247.4738464	1210.54	1293
db	SEDA Actor	2g	10	NA	8.474252849	1163.753333	33
db	SEDA Actor	100m	300	NA	250.5738879	1196.046667	1296
db	SEDA Actor	100m	10	NA	8.622383002	1143.963333	32
db	SEDA Queue	2g	300	NA	225.751635	1327.736667	1083.8
db	SEDA Queue	2g	10	NA	8.401889501	1160.8	37
db	SEDA Queue	100m	300	NA	229.705122	1303.386667	1388
db	SEDA Queue	100m	10	NA	8.601418007	1136.336667	34
db	NIO2	2g	300	NA	149.7219324	1976.773333	1127
db	NIO2	2g	10	NA	7.372583664	1335.7	26
db	NIO2	100m	300	NA	182.0857958	1639.706667	1246
db	NIO2	100m	10	NA	7.509059137	1310.463333	25
db	NIO2 Actor	2g	300	NA	1043.713581	287.33	1156
db	NIO2 Actor	2g	10	NA	34.7034771	287.1166667	36
db	NIO2 Actor	100m	300	NA	1061.39425	282.5366667	1106
db	NIO2 Actor	100m	10	NA	34.59198416	287.93	43